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++;
```

```
            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++) {
```

```
99          int numSimplices;
100         printf("Inserisci il numero di %d-simplessi: ", dim);
101         if (scanf("%d", &numSimplices) != 1) {
102             fprintf(stderr, "Input non valido.\n");
103             exit(1);
104         }
105         for (s = 0; s < numSimplices; s++) {
106             int m = dim + 1;
107             int* vertices = (int*)malloc(m * sizeof(int));
108             if (vertices == NULL) {
109                 fprintf(stderr, "Errore di allocazione della memoria.\n");
110                 exit(1);
111             }
112             printf("Inserisci gli elementi del %d-simplesso %d: ", dim, s + 1);
113             for (j = 0; j < m; j++) {
114                 if (scanf("%d", &vertices[j]) != 1) {
115                     fprintf(stderr, "Input non valido.\n");
116                     exit(1);
117                 }
118             }
119             // Ordinamento degli elementi per il corretto funzionamento di isSubset
120             qsort(vertices, m, sizeof(int), cmpInt);
121             addSimplex(&complexes[dim], vertices, m);
122         }
123     }
124
125     // Processo per trovare i simplessi massimali.
126     // Per ogni simplesso di dimensione inferiore a k, si verifica se è contenuto
        in un simplesso di dimensione superiore.
127     for (dim = k - 1; dim >= 0; dim--) {
128         simplex* newList = NULL;
129         int newSize = 0;
130         simplex* current = complexes[dim].simplices;
131         while (current != NULL) {
132             bool isMaximal = true;
133             for (supDim = dim + 1; supDim <= k && isMaximal; supDim++) {
134                 simplex* candidate = complexes[supDim].simplices;
135                 while (candidate != NULL && isMaximal) {
136                     if (isSubset(current, candidate)) {
137                         isMaximal = false;
138                     }
139                     candidate = candidate->next;
140                 }
141             }
142             simplex* nextNode = current->next;
143             if (isMaximal) {
144                 current->next = newList;
145                 newList = current;
146                 newSize++;
147             } else {
148                 free(current->vertices);
149                 free(current);
150             }
151             current = nextNode;
152         }
153         complexes[dim].simplices = newList;
154         complexes[dim].size = newSize;
155     }
156
157     // Output dei complessi massimali
158     printf("I complessi massimali sono:\n");
159     for (dim = 0; dim <= k; dim++) {
160         if (complexes[dim].simplices != NULL) {
161             printf("%d-simplessi massimali:\n", dim);
162             simplex* cur = complexes[dim].simplices;
163             while (cur != NULL) {
164                 printf("{");
165                 for (j = 0; j < cur->position; j++) {
166                     if (j > 0) {
167                         printf(", ");
```

```
                    }
                    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

```
#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)
```

```
41          return a->position - b->position;
42      for(i = 0; i < a->position; i++) {
43          if(a->vertices[i] != b->vertices[i])
44              return a->vertices[i] - b->vertices[i];
45      }
46      return 0;
47  }
48
49  // Rimuove i duplicati all'interno di un SimplicialComplex (lista concatenata)
50  void removeDuplicates(SimplicialComplex* complex) {
51      simplex *current, *runner, *prevRunner, *temp;
52      current = complex->simplices;
53      while(current != NULL) {
54          prevRunner = current;
55          runner = current->next;
56          while(runner != NULL) {
57              if(compareSimplex(current, runner) == 0) {
58                  prevRunner->next = runner->next;
59                  temp = runner;
60                  runner = runner->next;
61                  free(temp->vertices);
62                  free(temp);
63                  complex->size--;
64              } else {
65                  prevRunner = runner;
66                  runner = runner->next;
67              }
68          }
69          current = current->next;
70      }
71  }
72
73  // Genera tutti i sottoinsiemi non vuoti di un simplesso massimale e li aggiunge
74  // alla struttura dei simplessi "allComplexes" in base alla loro dimensione.
75  void generateSubsetsForSimplex(const simplex* maxSimplex, SimplicialComplex*
        allComplexes) {
76      int n = maxSimplex->position;
77      int total = 1 << n; // 2^n
78      int i, j;
79      for(i = 1; i < total; i++) {
80          int count = 0;
81          for(j = 0; j < n; j++) {
82              if(i & (1 << j))
83                  count++;
84          }
85          int* subset = (int*)malloc(count * sizeof(int));
86          if(subset == NULL) {
87              fprintf(stderr, "Errore di allocazione della memoria.\n");
88              exit(1);
89          }
90          int index = 0;
91          for(j = 0; j < n; j++) {
92              if(i & (1 << j)) {
93                  subset[index] = maxSimplex->vertices[j];
94                  index++;
95              }
96          }
97          // La dimensione del simplesso è count-1 (numero di vertici - 1)
98          int subsetDim = count - 1;
99          addSimplex(&allComplexes[subsetDim], subset, count);
100     }
101 }
102
103 // Libera tutta la memoria allocata per un SimplicialComplex
104 void freeSimplicialComplex(SimplicialComplex* complex) {
105     simplex* cur = complex->simplices;
106     simplex* next;
107     while(cur != NULL) {
108         next = cur->next;
109         free(cur->vertices);
```

```
110         free(cur);
111         cur = next;
112     }
113     complex->simplices = NULL;
114     complex->size = 0;
115 }
116 
117 int main(void) {
118     int k;
119     printf("Inserisci il numero massimo di dimensioni (k): ");
120     if(scanf("%d", &k) != 1) {
121         fprintf(stderr, "Input non valido.\n");
122         return 1;
123     }
124 
125     // Allocazione dinamica di due array di SimplicialComplex (uno per i simplessi
        massimali e uno per tutti quelli generati)
126     SimplicialComplex* maxSimplicialComplexes = (SimplicialComplex*)malloc((k + 1)
        * sizeof(SimplicialComplex));
127     SimplicialComplex* allSimplicialComplexes = (SimplicialComplex*)malloc((k + 1)
        * sizeof(SimplicialComplex));
128     if(maxSimplicialComplexes == NULL || allSimplicialComplexes == NULL) {
129         fprintf(stderr, "Errore di allocazione della memoria.\n");
130         exit(1);
131     }
132 
133     int dim, i, j;
134     // Inizializzazione delle strutture per ogni dimensione
135     for(dim = 0; dim <= k; dim++) {
136         maxSimplicialComplexes[dim].simplices = NULL;
137         maxSimplicialComplexes[dim].size = 0;
138         allSimplicialComplexes[dim].simplices = NULL;
139         allSimplicialComplexes[dim].size = 0;
140     }
141 
142     // Input dei simplessi massimali per ogni dimensione
143     for(dim = 0; dim <= k; dim++) {
144         int numMax;
145         printf("Inserisci il numero di %d-simplessi massimali: ", dim);
146         if(scanf("%d", &numMax) != 1) {
147             fprintf(stderr, "Input non valido.\n");
148             exit(1);
149         }
150         for(i = 0; i < numMax; i++) {
151             int m = dim + 1;
152             int* vertices = (int*)malloc(m * sizeof(int));
153             if(vertices == NULL) {
154                 fprintf(stderr, "Errore di allocazione della memoria.\n");
155                 exit(1);
156             }
157             printf("Inserisci gli elementi del %d-simplesso massimale %d: ", dim, i
                    + 1);
158             for(j = 0; j < m; j++) {
159                 if(scanf("%d", &vertices[j]) != 1) {
160                     fprintf(stderr, "Input non valido.\n");
161                     exit(1);
162                 }
163             }
164             qsort(vertices, m, sizeof(int), cmpInt);
165             addSimplex(&maxSimplicialComplexes[dim], vertices, m);
166         }
167     }
168 
169     // Per ogni simplesso massimale, genera tutti i suoi sottoinsiemi
170     simplex* current;
171     for(dim = 0; dim <= k; dim++) {
172         current = maxSimplicialComplexes[dim].simplices;
173         while(current != NULL) {
174             generateSubsetsForSimplex(current, allSimplicialComplexes);
175             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) {
```

```
16      SimplicialComplex* complex = (SimplicialComplex*) malloc((n)*sizeof(
            SimplicialComplex));
17      int i;
18
19      // aggiungo i vertici
20      complex[0].size=n;
21      Simplex *current, *new_simplex; // current sarà un puntatore all'ultimo
            elemento della lista degli 0-simplessi, new_simplex servirà per creare
            quelli da aggiungere
22
23      new_simplex = (Simplex*) malloc(sizeof(Simplex));
24      new_simplex->next = NULL;
25      new_simplex->position = 0;
26      new_simplex->vertices = (int*) malloc(sizeof(int));
27      new_simplex->vertices[0]=0;
28      complex[0].simplices=new_simplex;
29      current=new_simplex;
30
31      for (i=1; i<n; i++) {
32          new_simplex = (Simplex*) malloc(sizeof(Simplex));
33          new_simplex->next = NULL;
34          new_simplex->position = i;
35          new_simplex->vertices = (int*) malloc(sizeof(int));
36          new_simplex->vertices[0]=i;
37          current->next=new_simplex;
38          current=current->next;
39      }
40
41      // costruisco fino al livello n
42      for (i=1; i<n; i++) {
43          if (complex[i-1].simplices!=NULL) { // posso costruire i-simplessi solo se
                esistono degli (i-1)-simplessi
44              add_k_simplices_adjacency_matrix(complex,M,n,i);
45          }
46          else {
47              complex[i].simplices=NULL;
48              complex[i].size=0;
49          }
50      }
51
52      return complex;
53  }
54
55  // Costruisce il complesso simpliciale a partire dagli 1-simplessi troncato fino ai
        k-simplessi (inclusi), utilizzando la matrice di adiacenza.
56  // I vertici sono nominati da 0 a n-1, come gli indici della matrice.
57  // Utile ad esempio se serve calcolare il gruppo di omologia H_(k-1) e non serve l'
        intero complesso simpliciale.
58  SimplicialComplex* complex_from_adjacency_matrix_truncated (int** M, int n, int k)
        {
59      SimplicialComplex* complex = (SimplicialComplex*) malloc((k+1)*sizeof(
            SimplicialComplex));
60      int i;
61
62      // aggiungo i vertici
63      complex[0].size=n;
64      Simplex *current, *new_simplex; // current sarà un puntatore all'ultimo
            elemento della lista degli 0-simplessi, new_simplex servirà per creare
            quelli da aggiungere
65
66      new_simplex = (Simplex*) malloc(sizeof(Simplex));
67      new_simplex->next = NULL;
68      new_simplex->position = 0;
69      new_simplex->vertices = (int*) malloc(sizeof(int));
70      new_simplex->vertices[0]=0;
71      complex[0].simplices=new_simplex;
72      current=new_simplex;
73
74      for (i=1; i<n; i++) {
75          new_simplex = (Simplex*) malloc(sizeof(Simplex));
```

```
76          new_simplex->next = NULL;
77          new_simplex->position = i;
78          new_simplex->vertices = (int*) malloc(sizeof(int));
79          new_simplex->vertices[0]=i;
80          current->next=new_simplex;
81          current=current->next;
82      }
83
84      // costruisco fino al livello k
85      for (i=1; i<=k; i++) {
86          if (complex[i-1].simplices!=NULL) { // posso costruire i-simplessi solo se
                esistono degli (i-1)-simplessi
87              add_k_simplices_adjacency_matrix(complex,M,n,i);
88          }
89          else {
90              complex[i].simplices=NULL;
91              complex[i].size=0;
92          }
93      }
94
95      return complex;
96  }
97
98  // Costruisce il complesso simpliciale completo a partire dagli n vertici e dagli
        1-simplessi.
99  // I vertici verranno rinominati a partire da 0 e con interi consecutivi (se
        vogliamo posso scrivere una funzione per farli ritornare ai nomi originali)
100 SimplicialComplex* complex_from_1_simplices_complete (SimplicialComplex* complex,
        int n) {
101     int i,j;
102     int** M = (int**) malloc(n*sizeof(int*));
103     for (i=0; i<n; i++) M[i] = (int*) malloc(n*sizeof(int));
104     // salvo per comodita' in un array i vertici (per ovviare a problemi di
            nomenclatura ed evitare di scorrere sempre la lista)
105     int* vertices = (int*) malloc(n*sizeof(int));
106     Simplex* current=complex[0].simplices;
107     i=0;
108     while (current!=NULL) {
109         vertices[i]=current->vertices[0];
110         i++;
111         current=current->next;
112     }
113
114     // costruisco la matrice di adiacenza
115     for (i=0; i<n; i++) {
116         M[i][i]=1;
117         for (j=i+1; j<n; j++) {
118             // controllo se il vertice i-esimo e il j-esimo sono collegati da un 1-
                    simplesso
119             if (check_neighbor(complex,vertices[i],vertices[j])!=0) {
120                 M[i][j]=1; M[j][i]=1;
121             }
122             else {
123                 M[i][j]=0; M[j][i]=0;
124             }
125         }
126     }
127
128     return complex_from_adjacency_matrix_complete(M,n);
129 }
130
131 // Costruisce il complesso simpliciale a partire dagli n vertici e dagli 1-
        simplessi troncato fino ai k-simplessi (inclusi).
132 // I vertici verranno rinominati a partire da 0 e con interi consecutivi (se
        vogliamo posso scrivere una funzione per farli ritornare ai nomi originali)
133 SimplicialComplex* complex_from_1_simplices_truncated (SimplicialComplex* complex,
        int n, int k) {
134     int i,j;
135     int** M = (int**) malloc(n*sizeof(int*));
136     for (i=0; i<n; i++) M[i] = (int*) malloc(n*sizeof(int));
```

```
137     // salvo per comodita' in un array i vertici (per ovviare a problemi di
            nomenclatura ed evitare di scorrere sempre la lista)
138     int* vertices = (int*) malloc(n*sizeof(int));
139     Simplex* current=complex[0].simplices;
140     i=0;
141     while (current!=NULL) {
142         vertices[i]=current->vertices[0];
143         i++;
144         current=current->next;
145     }
146 
147     // costruisco la matrice di adiacenza
148     for (i=0; i<n; i++) {
149         M[i][i]=1;
150         for (j=i+1; j<n; j++) {
151             // controllo se il vertice i-esimo e il j-esimo sono collegati da un 1-
                    simplesso
152             if (check_neighbor(complex,vertices[i],vertices[j])!=0) {
153                 M[i][j]=1; M[j][i]=1;
154             }
155             else {
156                 M[i][j]=0; M[j][i]=0;
157             }
158         }
159     }
160 
161     return complex_from_adjacency_matrix_truncated(M,n,k);
162 }
163 
164 // Aggiunge al complesso simpliciale (che contiene fino ai (k-1)-simplessi) i k-
        simplessi specificati dalla matrice di adiacenza M n*n.
165 // 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).
166 void add_k_simplices_adjacency_matrix (SimplicialComplex* complex, int** M, int n,
        int k) {
167     // parto con 0 k-simplessi: la lista complex[k].simplices e' vuota
168     complex[k].simplices=NULL;
169     int num_k_simplices=0;
170     int i,j;
171 
172     // ricerco i k-simplessi da aggiungere:
173 
174     Simplex* previous_simplex = complex[k-1].simplices; // scorrerà i (k-1)-
            simplessi
175     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
176     while (previous_simplex!=NULL) { // per costruire i k-simplessi devo partire
            dai (k-1)-simplessi
177 
178         // cerco nuovi vertici a partire da quelli successivi all'ultimo gia'
                presente (funziona se il complesso e' ordinato)
179         for (j=previous_simplex->vertices[k-1]+1; j<n; j++) {
180             if (check_neighbor_adjacency_matrix(M,previous_simplex->vertices,k,j)
                    !=0) {
181                 // ho trovato il vicino j: costruisco il nuovo k-simplesso
182 
183                 if (num_k_simplices==0) { // caso in cui aggiungo il primo k-
                        simplesso alla lista
184                     new_simplex = (Simplex*) malloc(sizeof(Simplex));
185                     new_simplex->next = NULL;
186                     new_simplex->position = num_k_simplices;
187                     new_simplex->vertices = (int*) malloc((k+1)*sizeof(int));
188                     for (i=0; i<k; i++) new_simplex->vertices[i]=previous_simplex->
                            vertices[i];
189                     new_simplex->vertices[k]=j;
190 
191                     complex[k].simplices=new_simplex;
192                     current=new_simplex;
193                 }
```

```
                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) {
```

```
65                      FILE* f = fopen(file, "rt");
66                      if (!f) {
67                              printf("Errore nell'apertura del file\n");
68                              return NULL;
69                      }
70              else
71                              printf("File aperto correttamente\n");
72                      int size = 0;
73                      fscanf(f, "%d", &size);
74                      printf("size = %d\n", size);
75                      SimplicialComplex* complex = (SimplicialComplex*)malloc(
                            size * sizeof(SimplicialComplex));
76                      int n = 0;
77                      for (int i = 0; i < size; i++) {
78                              fscanf(f, "%d", &n);
79                              complex[i].size = n;
80                              complex[i].simplices = createSimplex_file(i, n, f);
81                      }
82                      fclose(f);
83                      return complex;
84              }
85
86      // funzione per stampare un simplesso
87      void printComplex(SimplicialComplex* complex, int size){
88              Simplex* app;
89                      for (int i = 0; i < size; i++) {
90                                      printf("%d-simplessi\n", i);
91                                      printf("size = %d\n", complex[i].size);
92                              app = complex[i].simplices;
93                  if (app) {
94                      printf("{ ");
95                      while (app) {
96                          printf("( ");
97                          for (int j = 0; j <= i; j++)
98                              printf("%d ", app->vertices[j]);
99                          printf(")");
100                         app = app->next;
101                         if (app)
102                             printf(", ");
103                     }
104                     printf(" }\n");
105                 }
106                     }
107             return;
108     }
109
110     // funzione per creare un sottoinsieme di un simplesso togliendo il vertice
            i-esimo
111     int* creasubs(int* v, int i, int n) {
112         int* s = (int*)malloc((n) * sizeof(int));
113         int k = 0;
114             //porto il cilco fino a n perchè il vertice i-esimo non deve essere
                    inserito quindi inserisco solo n-1 vertici
115         for (int j = 0; j <= n; j++) {
116             if (j != i) {
117                 s[k] = v[j];
118                 k++;
119             }
120         }
121         return s;
122     }
123
124     //controlla se due vettori sono uguali
125     bool equal(int* v, int* s, int n) {
126         for (int i = 0; i < n; i++) {
127             if (v[i] != s[i])
128                 return false;
129         }
130         return true;
131     }
```

```
132
133         //Funzione per controllare se i sottoinsiemi dell' n simplesso sono
                contenuti nell' n-1 simplesso
134         bool isIn(Simplex* s, Simplex* s1, int sizeN) {
135             int* v = (int*)malloc(sizeN * sizeof(int));
136             Simplex* Nsimp = s, * Nmosimp = s1;
137             bool isin = false;
138                 //fino a quando non ho terminato la lista degli n-simplessi
139             while (Nsimp) {
140                 for (int i = 0; i < sizeN + 1; i++) {
141                                 //creo i sottoinsiemi del n-simplesso
142                     v = creasubs(Nsimp->vertices, i, sizeN+1);
143                     isin = false;
144                                 //fino a quando non ho terminato la lista degli n
                                        -1-simplessi oppure ho trovato il sottoinsieme
145                     while (Nmosimp && !isin) {
146                                         //confronto i due vettori
147                         if (equal(Nmosimp->vertices, v, sizeN)) {
148                             isin = true;
149                         }
150                         else
151                             isin = false;
152                         Nmosimp = Nmosimp->next;
153                     }
154                     if (!isin)
155                         return false;
156                     Nmosimp = s1;
157                 }
158                 Nsimp = Nsimp->next;
159             }
160             return true;
161         }
162
163         // Funzione per controllare se il complesso è simpliciale
164         bool isSimplicial(SimplicialComplex* sc, int size) {
165                 //faccio il controllo per ogni n-simplesso partendo dall'ultimo
166             for (int i = size - 1; i > 0; i--)
167                 if (!isIn(sc[i].simplices, sc[i - 1].simplices, i))
168                     return false;
169             return true;
170         }
171
172         // Funzione per creare un simplesso sapendo prima le dimensioni
173         Simplex* createSimplex(int size, int n) {
174             Simplex* simplex, * head, * app;      // Definiamo la testa del
                    simplicio + due simplex ausiliari
175             simplex = (Simplex*)malloc(sizeof(Simplex));
176                 //mi salvo la testa del simplesso e la riempio
177             head = simplex;
178             simplex->next = NULL;
179             simplex->vertices = (int*)malloc((size + 1) * sizeof(int));
180             printf("Inserisci i %d-simplessi scrivendo i vertici in ordine
                    crescente separati da uno spazio es: (1 2 3)\n", size);
181             for (int j = 0; j < size + 1; j++) {
182                 scanf("%d", &simplex->vertices[j]);
183             }
184                 simplex->position = 0;
185                 //riempio i restanti n-1 simplessi utilizzando un simplex
                        ausiliario
186             for (int i = 1; i < n; i++) {
187                 app = (Simplex*)malloc(sizeof(Simplex));
188                 app->next = NULL;
189                 app->vertices = (int*)malloc((size + 1) * sizeof(int));
190                 for (int j = 0; j < size + 1; j++) {
191                     scanf("%d", &app->vertices[j]);
192                 }
193                         app->position = i;
194                 simplex->next = app;
195                 simplex = simplex->next;
196             }
```

```
197             return head;
198         }
199 
200         //stessa funzione di sopra ma leggo da file
201         Simplex* createSimplex_file(int size, int n, FILE* f) {
202             Simplex* simplex, * head, * app;
203             simplex = (Simplex*)malloc(sizeof(Simplex));
204             head = simplex;
205             simplex->next = NULL;
206             simplex->vertices = (int*)malloc((size + 1) * sizeof(int));
207             for (int j = 0; j < size + 1; j++) {
208                 fscanf(f, "%d", &simplex->vertices[j]);
209             }
210             simplex->position = 0;
211             for (int i = 1; i < n; i++) {
212                 app = (Simplex*)malloc(sizeof(Simplex));
213                 app->next = NULL;
214                 app->vertices = (int*)malloc((size + 1) * sizeof(int));
215                 for (int j = 0; j < size + 1; j++) {
216                     fscanf(f, "%d", &app->vertices[j]);
217                 }
218                 app->position = i;
219                 simplex->next = app;
220                 simplex = simplex->next;
221             }
222             return head;
223         }
224 
225     #pragma endregion
226 
227 
228     #pragma region edge_Matrix
229 
230                 // Funzione per calcolare la matrice di bordo
231                 int** edge_Matrix(SimplicialComplex* sc, int n) {
232             int i = 0, j = 0, k = -1, row = sc[n - 1].size, col = sc[n].size;
233             int** matrix;
234             matrix = (int**)calloc(row,sizeof(int*));
235                         Simplex* Nsimp, * Nmosimp;
236                         int* v = (int*)malloc(n * sizeof(int));
237                         Nsimp = sc[n].simplices;
238                         Nmosimp = sc[n - 1].simplices;
239                         //itero i vari n-simplessi
240             for (int i = 0; i < row; i++)
241                 matrix[i] = (int*)calloc(col, sizeof(int));
242                 for (int i = 0; i < sc[n].size; i++) {
243                                         //itero i vari n-1-simplessi
244                                         //creo i sotto inisiemi di ciascun simplesso e
                                                vedo che posizione ha nella base
245                     for (int j = 0; j <= n; j++) {
246                                                 v = creasubs(Nsimp->vertices, j, n);
247                                                 k = base_number(Nmosimp, v, n);
248                                                 //printf("k = %d\n", k);
249                         if (k != -1) {
250                             matrix[k][i] = pow(-1, j);
251                         }
252                     }
253                                         Nsimp = Nsimp->next;
254                         }
255                         return matrix;
256                 }
257 
258                 int base_number(Simplex* sc, int* v, int n) {
259                         Simplex* app = sc;
260                         while (app) {
261                                 if (equal(app->vertices, v, n))
262                                         return app->position;
263                                 app = app->next;
264                         }
265                         return -1;
```

```
266                 }
267     #pragma endregion
268
269     #pragma region matrix_rank
270         void swap_rows(int** matrix, int row1, int row2, int cols) {
271             for (int i = 0; i < cols; i++) {
272                 double temp = matrix[row1][i];
273                 matrix[row1][i] = matrix[row2][i];
274                 matrix[row2][i] = temp;
275             }
276         }
277
278                 /*
279         // Funzione per calcolare il rango di una matrice
280                 int matrix_rank(int** matrix, int rows, int cols) {
281             int rank = cols;  // Inizialmente assumiamo il rango massimo possibile
282
283             for (int row = 0; row < rank; row++) {
284                 // Controlliamo se l'elemento diagonale è diverso da zero
285                 if (fabs(matrix[row][row]) > 0) {
286                     // Eliminiamo gli elementi sotto l'elemento pivot
287                     for (int i = 0; i < rows; i++) {
288                         if (i != row) {
289                             double factor = matrix[i][row] / matrix[row][row];
290                             for (int j = 0; j < cols; j++) {
291                                 matrix[i][j] -= factor * matrix[row][j];
292                             }
293                         }
294                     }
295                 }
296                 else {
297                     // Se il pivot è zero, cerchiamo una riga non nulla da
                        scambiare
298                     int reduce = 1;
299                     for (int i = row + 1; i < rows; i++) {
300                         if (fabs(matrix[i][row]) > 0) {
301                             swap_rows(matrix, row, i, cols);
302                             reduce = 0;
303                             break;
304                         }
305                     }
306
307                     if (reduce) {
308                         // Se non troviamo una riga valida, riduciamo il rango
309                         rank--;
310
311                         // Spostiamo l'ultima colonna a sinistra
312                         for (int i = 0; i < rows; i++) {
313                             matrix[i][row] = matrix[i][rank];
314                         }
315                     }
316                     row--;
317                 }
318             }
319
320             return rank;
321                 }*/
322 #pragma endregion
323
324 #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 curve chiuse e indipendenti non piene H_2 il numero di superfici chiuse, indipendenti e non piene 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, 2);

        //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);
```

```
50          printf("Il gruppo di omologia H_%d del complesso simpliciale inserito ha
                dimensione = %d\n", n - 1, h);
51
52          //esempio di calcolo delll'imologia decomposta in z-moduli
53          int** h1 = NULL;
54          int** M = NULL;
55          M = input_null(M, 3, 2);
56          int** M2 = NULL;
57          M2 = input_null(M2, 1, 3);
58
59          M[0][0] = 8;
60          M[0][1] = -15;
61          M[1][0] = -15;
62          M[1][1] = 29;
63          M[2][0] = -6;
64          M[2][1] = 13;
65
66          M2[0][0] = -9;
67          M2[0][1] = -6;
68          M2[0][2] = 3;
69
70          h1 = omologia(M, 3, 2, M2, 1, 3);
71
72
73          return;
74  }
```

Listing 6: Omo.h

```
1   #pragma once
2   #ifndef Omo
3           #define Omo
4           #include "..\Include\Matrix.h"
5
6       // Questo file include funzioni per la manipolazione delle omologie
7
8           //Definizioni funzioni
9           int** omologia(int**, int, int, int**, int, int);
10          int compare(const void*, const void*);
11
12          int dim_omologia(int**, int, int, int**, int, int);
13          bool is_Complex(int**, int, int, int**, int, int);
14
15          //Implementazione
16
17          //semplcie funzione per comparare due interi
18          int compare(const void* a, const void* b) {
19                  return (*(int*)a - *(int*)b);
20          }
21
22          //funzione per controllare se due matrici formano un complesso ovvero se la
                loro moltiplicazione è nulla
23          //proprietà delle matrici di bordo
24          bool is_Complex(int** matrix1, int row1, int col1, int** matrix2, int row2,
                int col2) {
25                  if (col2 != row1) {
26                          return false;
27                  }
28                  int** temp = mul_matrix(matrix2, row2, col2, matrix1, row1, col1);
29                  //print_matrix(temp, row2, col1);
30                  for (int i = 0; i < row2; i++) {
31                          for (int j = 0; j < col1; j++) {
32                                  if (temp[i][j] != 0) {
33                                          return false;
34                                  }
35                          }
36                  }
```

```
37                  return true;
38          }
39
40
41          //funzione per calcolare la dimensione dell'omologia n richiesta passo in
                input solo le due matrici di bordo
42          int dim_omologia(int** matrix1, int row1, int col1, int** matrix2, int row2
                , int col2) {
43                  int h = 0, ck = 0, rk = 0, rk1 = 0;
44
45                  //controlliamo che le due matrici diano un complesso
46                  if (matrix1 && matrix2) {
47                          if (!is_Complex(matrix1, row1, col1, matrix2, row2, col2))
                                {
48                                  return -1;
49                          }
50                  }
51
52                  int** D = NULL;
53                  int** S = NULL;
54                  int** T = NULL;
55
56                  //calcoliamo la dimensione dell'omologia come h = ck - rk - rk1
                        dove ck è il numero di n simplessi
57                  //rk è il rango della matrice di bordo n+1-esima e rk1 è il rango
                        della matrice di bordo n-esima
58                  //controlliamo se esistono le matrici e quindi ne calcolo il rango
                        utilizzando la matrice di smith
59                  if (matrix2) {
60
61                          D = input_null(D, row2, col2);
62                          S = input_id(S, row2);
63                          T = input_id(T, col2);
64
65                          for (int i = 0; i < row2; i++) {
66                                  for (int j = 0; j < col2; j++) {
67                                          D[i][j] = matrix2[i][j];
68                                  }
69                          }
70
71
72                          ck = col2;
73
74                          SmithNormalForm(D, S, T, row2, col2);
75                          rk = rank_matrix_diag(D, row2, col2);
76                          printf("rk = %d\n", rk);
77
78                          free(D);
79                          free(S);
80                          free(T);
81                  }
82                  else
83                          rk = 0;
84
85                  if (matrix1) {
86                          D = input_null(D, row1, col1);
87                          S = input_id(S, row1);
88                          T = input_id(T, col1);
89
90                          for (int i = 0; i < row1; i++) {
91                                  for (int j = 0; j < col1; j++) {
92                                          D[i][j] = matrix1[i][j];
93                                  }
94                          }
95
96                          ck = row1;
97
98                          SmithNormalForm(D, S, T, row1, col1);
99                          rk1 = rank_matrix_diag(D, row1, col1);
100                         printf("rk1 = %d\n", rk1);
```

```
101
102                     free(D);
103                     free(S);
104                     free(T);
105             }
106             else
107                     rk1 = 0;
108
109
110             printf("ck = %d\n", ck);
111             h = ck - rk - rk1;
112
113             return h;
114     }
115
116
117     //Come rappresentare l'omologia in z-moduli
118     int** omologia(int** matrix1, int row1, int col1, int** matrix2, int row2,
            int col2) {
119             int** h = NULL;
120
121             //controlliamo che le due matrici diano un compleso
122             if (!is_Complex(matrix1, row1, col1, matrix2, row2, col2)) {
123                     printf("Errore: le matrici non formano un complesso\n");
124                     return NULL;
125             }
126
127             int** D = NULL;
128             D = input_null(D, row2, col2);
129             int** S = NULL;
130             S = input_id(S, row2);
131             int** T = NULL;
132             T = input_id(T, col2);
133
134
135             for (int i = 0; i < row2; i++) {
136                     for (int j = 0; j < col2; j++) {
137                             D[i][j] = matrix2[i][j];
138                     }
139             }
140
141             //calcoliamo la forma di smith della matrice An
142             SmithNormalForm(D, S, T, row2, col2);
143
144             //troviamo il rango della matrice ed estraiamo una base del nucleo
                    ovvero
145             //le ultime r colonne della matrice T (SAT = D) che salvo in T2
146             int r = rank_matrix_diag(D, row2, col2);
147
148             int** T2 = NULL;
149             T2 = input_null(T2, col2, col2-r);
150
151
152             for (int i = 0; i < col2; i++) {
153                     for (int j = 0; j < col2 - r; j++) {
154                             T2[i][j] = T[i][j + r];
155
156                     }
157             }
158
159             //troviamo la forma di smith della base del nucleo appena trovata
160             //per risolvere Bx = An+1 negli interi moltiplicando per l'inverso
                    di ogni matrice che gia ho
161             //e fermandomi al rango della matrice B
162
163             int** S1 = NULL;
164             S1 = input_id(S1, col2);
165             int** T1 = NULL;
166             T1 = input_id(T1, col2 - r);
167             int** D1 = NULL;
```

```
168                 D1 = input_null(D1, col2, col2 - r);
169
170                 for (int i = 0; i < col2; i++) {
171                         for (int j = 0; j < col2 - r; j++) {
172                                 D1[i][j] = T2[i][j];
173                         }
174                 }
175
176                 SmithNormalForm(D1, S1, T1, col2, col2 - r);
177                 int** temp = NULL;
178
179                 //mi fermo a col2-r righe per s1*A
180                 temp = mul_matrix(S1, col2 - r, col2, matrix1, row1, col1);
181
182                 //D^-1 * S1 * A
183                 for (int i = 0; i < col2 - r; i++)
184                 {
185                         for (int j = 0; j < col1; j++)
186                         {
187                                 temp[i][j] = temp[i][j] / D1[i][i];
188                         }
189                 }
190
191                 //T1 * D^-1 * S1 * A=x
192                 temp = mul_matrix(T1, col2 - r, col2 - r, temp, col2 - r, col1);
193
194
195                 free(S1);
196                 free(T1);
197                 free(D1);
198
199                 //ora decompongo in z-moduli il risultato x=temp sempre attaverso
                        smith e la matrice diagonale
200
201                 S1 = NULL;
202                 S1 = input_id(S1, col2 - r);
203                 T1 = NULL;
204                 T1 = input_id(T1, col2 - r);
205                 D1 = NULL;
206                 D1 = input_null(D1, col2 - r, col2 - r);
207
208                 for (int i = 0; i < col2 - r; i++) {
209                         for (int j = 0; j < col2 - r; j++) {
210                                 D1[i][j] = temp[i][j];
211                         }
212                 }
213
214                 SmithNormalForm(D1, S1, T1, col2 - r, col2 - r);
215                 h = (int*)calloc(col2 - r - 1, sizeof(int));
216                 for (int i = 0; i < col2 - r ; i++) {
217                         h[i] = D1[i][i];
218                 }
219
220                 //printiamo il risultato ottenuto ovvero i valori di h
221                 printf("\nDecomposizione di h\n");
222                 printf("Z_%d", h[0]);
223                 for (int i = 1; i < col2 - r; i++) {
224                         printf(" + ");
225                         printf("Z_%d", h[i]);
226                 }
227
228                 /* non sempre trovo un minore invertibile negli itneri quindi
                        questo codice non va bene
229                 int** B = NULL;
230                 int** B_inv = NULL;
231                 int** C = NULL;
232                 B = input_null(B, col2 - r, col2 - r);
233                 B_inv = input_null(B_inv, col2 - r, col2 - r);
234                 C = input_null(C, col2 - r, col1);
235
```

```
236                 //cerco un minore invertibile di T2
237                 //uso gauss per mettere la matrice in forma diagonale
238                 //poi prendo gli indici con elemnti non nulli e li salvo per
                        estrarne un minore
239                 int* index = (int*)calloc(col2 - r, sizeof(int));
240                 gauss_rectangular(T2, col2, col2 - r);
241                 printf("Matrice T2 gauss\n");
242                 print_matrix(T2, col2, col2 - r);
243                 for (int i = 0; i < col2 - r; i++) {
244                         int k = 0;
245                         for (k = 0; k < col2; k++) {
246                                 if (T2[k][i] != 0) {
247                                         index[i] = k;
248
249                                         for (int j = i; j < col2 - r; j++) T2[k][j]
                                                = 0;
250                                         break;
251                                 }
252                         }
253                 }
254                 //ordino gli indici per prendere le righe corrispondenti
255                 //sia su B che su A
256                 qsort(index, col2 - r, sizeof(int), compare);
257
258
259                 for (int i = 0; i < col2 - r; i++) {
260                         printf("i = %d \n", index[i]);
261                         int k = index[i];
262                         for (int j = 0; j < col2 - r; j++) {
263                                 B[i][j] = T[k][j + r];
264                         }
265
266                         for( int j = 0; j < col1; j++){
267                                 C[i][j] = matrix1[k][j];
268                         }
269                 }
270
271                 int iter = -1;
272
273                 //cerco il minore invertibile di B negli interi condizione sul
                        determinante
274                 while (det_matrix_triangular(B, col2 - r) != 1 &&
                        det_matrix_triangular(B, col2 - r) != - 1) {
275                         iter++;
276                         if (iter == col2)
277                         {
278                                 printf("Matrice non invertibile\n");
279                                 return NULL;
280                         }
281                         else
282                         {
283
284                                 while(T[iter][col2 - 1] == 0 && iter < col2)
285                                         iter++;
286                         }
287
288                         for (int j = 0; j < col2 - r; j++) {
289                                 B[col2 - r - 1][j] = T[iter][j + r];
290                         }
291
292                         for (int j = 0; j < col1; j++) {
293                                 C[col2 - r - 1][j] = matrix1[iter][j];
294                         }
295                 }
296
297
298
299                 //inverto la matrice
300                 B_inv = invert_matrix_integer(B, col2 - r);
301
```

```
302                 printf("Matrice B\n");
303                 print_matrix(B, col2 - r, col2 - r);
304                 printf("Matrice B_inv\n");
305                 print_matrix(B_inv, col2 - r, col2 - r);
306 
307                 int** temp = mul_matrix(B_inv, col2 - r, col2 - r, C, col2 - r,
                        col2 - r);
308 
309 
310                 printf("Matrice molt\n");
311                 print_matrix(temp, col2 - r, col2 - r);
312 
313 
314 
315                 printf("Matrice C\n");
316                 print_matrix(C, col2 - r, col1);
317                 */
318 
319 
320                 free(D);
321                 free(S);
322                 free(T);
323                 free(S1);
324                 free(T1);
325                 free(D1);
326                 free(T2);
327 
328                 return h;
329         }
330 
331 
332 #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

```
1  #pragma once
2  #ifndef Smith
3  #define Smith
4 
5  int** input_sm (int**, int**, int, int);
6  int** input_id (int**, int);
7  int** input_null (int**, int, int);
8  void printMatrix (int**, int, int, int);
9  bool nullMatrix (int**, int, int);
10 int** traspMatrix (int**, int**, int, int);
11 void switchZeros (int**, int**, int**, int**, int, int, int);
12 void switchSigns (int**, int**, int, int, int, int);
```

```
13 void swapRows (int**, int, int, int);
14 void addRows (int**, int, int, int, int);
15 void minUp (int**, int**, int, int, int, int);
16 void div_rem (int**, int**, int, int, int, int);
17 bool nullVector (int**, int, int);
18 void Smith_fst (int**, int**, int**, int**, int**, int, int, int);
19 int my_min (int, int);
20 bool bool_dk (int**, int, int);
21 void SmithNormalForm (int**, int**, int**, int, int);
22 void multiplyMatrices(int**, int**, int**, int, int, int);
23 void Smith_D (int**, int, int);
24 int checkDiagZeros (int**, int);
25 void Smith_fst5mat (int**, int**, int**, int**, int**, int**, int**, int**, int,
       int, int);
26 void SmithNormalForm5mat (int**, int**, int**, int**, int**, int, int);
27 void minUp5mat (int**, int**, int**, int, int, int, int);
28 void div_rem5mat (int**, int**, int**, int, int, int, int);
29 void switchSigns5mat (int**, int**, int**, int, int, int, int);
30 void switchZeros5mat (int**, int**, int**, int**, int**, int, int, int);
31
32 // inizializzazione matrice i cui elementi sono scelti dall'utente
33 int** input_sm (int **mat, int **temp, int m, int n) {
34         mat=calloc(m, sizeof(int*));
35         for (int i=0; i<m; i++) {
36                 mat[i]=calloc(n, sizeof(int));
37         }
38         printf("Elementi della matrice:\n");
39         for (int i=0; i<m; i++) {
40                 for (int j=0; j<n; j++) {
41                         scanf("%d", &mat[i][j]);
42                         temp[i][j]=mat[i][j];
43                 }
44         }
45         return mat;
46 }
47
48 // inizializzazione matrice identità
49 int** input_id (int **mat, int m) {
50         mat=calloc(m, sizeof(int *));
51         for (int i=0; i<m; i++) {
52                 mat[i]=calloc(m, sizeof(int));
53         }
54         for (int i=0; i<m; i++) {
55                 mat[i][i]=1;
56         }
57         return mat;
58 }
59
60 // inizializzazione matrice nulla
61 int** input_null(int **mat, int m, int n) {
62         mat=calloc(m, sizeof(int*));
63         for (int i=0; i<m; i++) {
64                 mat[i]=calloc(n, sizeof(int));
65         }
66         return mat;
67 }
68
69 // stampare la matrice
70 void printMatrix (int **mat, int m, int n, int t) {
71         for (int i=0; i<m; i++) {
72                 for (int j=0; j<n; j++) {
73                         if (mat[i][j]<0) {
74                                 printf("%d ", mat[i][j]);
75                         } else {
76                                 printf(" %d ", mat[i][j]);
77                         }
78                 }
79                 printf("\n");
80                 if (i!=(m-1)) {
81                         for (int l=0; l<t; l++) {
```

```
82                                      printf(" ");
83                              }
84                      }
85              }
86              return;
87      }
88
89      // verificare che la matrice non sia nulla
90      bool nullMatrix (int **mat, int m, int n) {
91              for (int i=0; i<m; i++) {
92                      for (int j=0; j<n; j++) {
93                              if (mat[i][j]!=0) {
94                                      return false;
95                              }
96                      }
97              }
98              return true;
99      }
100
101     // trasporre una matrice
102     int** traspMatrix (int **mat, int **trasp, int m, int n) {
103             for (int i=0; i<n; i++) {
104                     for (int j=0; j<m; j++) {
105                             trasp[i][j]=mat[j][i];
106                     }
107             }
108             return trasp;
109     }
110
111     // spostare un'eventuale colonna nulla in fondo alla matrice
112     void switchZeros (int **A, int **T, int **At, int **Tt, int m, int n, int k) {
113             bool var=true;
114             for (int i=k; i<m; i++) {
115                     if (A[i][k]!=0) {
116                             var=false;
117                             break;
118                     }
119             }
120             if (var) {
121                     At=traspMatrix(A, At, m, n); // swapRows lavora sulle righe, quindi
                            per invertire le colonne bisogna lavorare sulla trasposta
122                     Tt=traspMatrix(T, Tt, n, n); // invertire le colonne è un'
                            operazione che lavora su T
123                     for (int j=k; j<(n-1); j++) {
124                             swapRows(At, j, j+1, m); // inverte le righe j e j+1 di una
                                    matrice
125                             swapRows(Tt, j, j+1, n);
126                     }
127                     A=traspMatrix(At, A, n, m); // trasposta di trasposta per tornare
                            nella forma originaria
128                     T=traspMatrix(Tt, T, n, n);
129             }
130             return;
131     }
132
133     // invertire i segni di una riga della matrice
134     void switchSigns (int **A, int **mat, int c, int m, int n, int k) {
135             for (int i=k; i<m; i++) {
136                     if (A[i][k]<0) {
137                             for (int j=k; j<n; j++) {
138                                     A[i][j]=-A[i][j]; // matrice A
139                             }
140                             for (int l=0; l<c; l++) {
141                                     mat[i][l]=-mat[i][l]; // matrice S o T a seconda
                                            dei casi
142                             }
143                     }
144             }
145             return;
146     }
```

```
147
148 // scambaire le righe row1 e row2 di una matrice
149 void swapRows (int **mat, int row1, int row2, int n) {
150         int temp;
151         for (int i=0; i<n; i++) {
152                 temp=mat[row1][i];
153                 mat[row1][i]=mat[row2][i];
154                 mat[row2][i]=temp;
155         }
156         return;
157 }
158
159 // aggiungere alla riga row2 un multiplo (mult) della riga row1
160 void addRows (int **mat, int row1, int row2, int mult, int n) {
161         for (int j=0; j<n; j++) {
162                 mat[row2][j]=mat[row2][j]-mult*mat[row1][j];
163         }
164         return;
165 }
166
167 // portare il minimo della colonna k di una matrice in posizione [k][k]
168 void minUp (int **A, int **mat, int c, int m, int n, int k) {
169         int min=A[k][k], r_min=k;
170         for(int i=k; i<m; i++) {
171                 if (A[i][k]<min && A[i][k]!=0 || min==0) {
172                         min=A[i][k]; // individuare il minimo
173                         r_min=i; // individuare la riga del minimo
174                 }
175         }
176         swapRows(A, k, r_min, n); // invertire la riga k e r_min della matrice per
              portare il minimo in alto
177         swapRows(mat, k, r_min, c); // stesso lavoro anche su S o T a seconda dei
              casi
178         return;
179 }
180
181 // divisione con i resti
182 void div_rem (int **A, int **mat, int c, int m, int n, int k) {
183         int quot=0;
184         for (int i=k+1; i<m; i++) {
185                 quot=A[i][k]/A[k][k]; // individuare il quzionte della divisione
186                 addRows(A, k, i, quot, n); // modificare le righe di A (inserendo i
                      resti)
187                 addRows(mat, k, i, quot, c); // stesso lavoro su S o T a seconda
                    dei casi
188         }
189         return;
190 }
191
192 // verificare se la colonna è della forma [0, ..., 0, d, 0, ..., 0] con d in
      posizione k
193 bool nullVector (int **mat, int m, int k) {
194         for (int i=0; i<m; i++) {
195                 if (i!=k && mat[i][k]!=0) {
196                         return false;
197                 }
198         }
199         return true;
200 }
201
202 // prima forma di Smith che porta a un matrice diagonale ma i cui elementi in
      diagonale non necessariamente sono multipli
203 void Smith_fst (int **A, int **S, int **T, int **At, int **Tt, int m, int n, int k)
      {
204         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]
205                 switchZeros(A, T, At, Tt, m, n, k); // per prima cosa sposto l'
                      eventuale colonna nulla
206                 while (!nullVector(A, m, k)) { // comincio a lavorare sulle colonne
```

```
207                         switchSigns(A, S, m ,m, n, k); // cambio i segni della
                                colonna k
208                         while (!nullVector(A, m, k)) { // finchè la colonna non è
                                della forma corretta applico a ripetizione le seguenti
                                funzioni
209                                 minUp(A, S, m, m, n, k); // porto il minimo in alto
                                    
210                                 div_rem(A, S, m, m, n, k); // divisione con i resti
211                         }
212                 }
213                 // per poter lavorare sulle righe traspongo la matrice e ripeto lo
                        stesso procedimento fatto per la colonna
214                 At=traspMatrix(A, At, m, n);
215                 Tt=traspMatrix(T, Tt, n, n);
216                 while (!nullVector(At, n, k)) {
217                         switchSigns(At, Tt, n, n, m, k);
218                         while (!nullVector(At, n, k)) {
219                                 minUp(At, Tt, n, n, m, k);
220                                 div_rem(At, Tt, n, n, m, k);
221                         }
222                 }
223                 A=traspMatrix(At, A, n, m);
224                 T=traspMatrix(Tt, T, n, n);
225         }
226         switchSigns(A, S, m, m, n, k); // cambio il segno dell'ultimo elemento in
              diagonale nel caso sia negativo
227         return;
228 }
229 
230 // individuo il min tra due interi
231 int my_min (int m, int n) {
232         if (m<n) {
233                 return m;
234         } else {
235                 return n;
236         }
237 }
238 
239 // 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))
240 bool bool_dk (int **A, int r, int k) { // r=min(m, n) e k è l'indice dell'elemento
        diagonale che sto verificando
241         int quot=0, rem=0;
242         for (int i=k+1; i<r; i++) { // scorro i da k+1 ad r
243                 rem=A[i][i]%A[k][k];
244                 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
245                         return false;
246                 }
247         }
248         return true;
249 }
250 
251 // verifico che la diagonale non abbia zeri fuori posto
252 int checkDiagZeros (int **A, int r) {
253         for (int i=0; i<r; i++) {
254                 if (A[i][i]==0) { // appena trovo un elemento diagonale nullo
                        verifico i successivi
255                         for (int j=i; j<r; j++) {
256                                 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)
257                                         return i;
258                                 }
259                         }
260                         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)
```

```
261                 }
262         }
263         return r;
264 }
265
266 // forma normale di Smith
267 void SmithNormalForm (int **A, int **S, int **T, int m, int n) {
268         int **At=NULL;
269         At=input_null(At, n, m);
270         int **Tt=NULL;
271         Tt=input_null(Tt, n, n);
272
273         int r=my_min(m, n);
274
275         for (int k=0; k!=r; k++) {
276                 Smith_fst(A, S, T, At, Tt, m, n, k); // prima raggiungo la forma
                        diagonale
277         }
278
279         // verifico che la forma diagonale raggiunta abbia gli zeri tutti in fondo
                altrimenti riapplico smith_fst dopo aver fatto delle modifiche
280         int err=checkDiagZeros(A, r);
281         if (err!=r) { // se il check è fallito, ossia ci sono degli zeri fuori
                posto
282                 for (int j=err+1; j<r; j++) { // sommo alla riga err (con lo zero
                        fuori posto) le righe successive j
283                         addRows(A, j, err, -1, n);
284                         addRows(S, j, err, -1, m);
285                 }
286
287                 for (int k=err; k<r; k++) {
288                         Smith_fst(A, S, T, At, Tt, m, n, k); // prima raggiungo la
                                forma diagonale
289                 }
290         }
291
292         // una volta raggiunta una forma diagonale con gli zeri in fondo verifico
                che la forma sia effettivamente di Smith
293         for (int it=0; it<r; it++) {
294                 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)
295                         break; // quindi posso uscire dal for perchè la verifica è
                                completa
296                 }
297                 if (!bool_dk(A, r, it)) { // nel caso non sia di Smith
298                         for (int i=it+1; i<r; i++) { // sommo alla riga it tutte le
                                righe successive i
299                                 addRows(A, i, it, -1, n);
300                                 addRows(S, i, it, -1, m);
301                         }
302                         for (int iter=it; iter<r; iter++) {
303                                 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
304                         }
305                 }
306         }
307         switchSigns(A, S, m, m, n, r-1); // cambio eventualmente il segno dell'
                ultimo elemento in diagonale
308         return;
309 }
310
311 // moltiplicare due matrici: C=A*B
312 void multiplyMatrices(int **A, int **B, int **C, int m, int n, int p) {
313         // Itera sulle righe di A
314         for (int i=0; i<m; i++) {
315                 // Itera sulle colonne di B
316                 for (int j=0; j<p; j++) {
317                         C[i][j] = 0;
```

```
318                         // Calcola il prodotto scalare tra la riga i di A e la
                                colonna j di B
319                         for (int k=0; k<n; k++) {
320                                 C[i][j]=C[i][j]+A[i][k]*B[k][j];
321                         }
322                 }
323         }
324 }
325
326 // forma normale di Smith senza ricordare le matrici S e T
327 void Smith_D (int **A, int m, int n) {
328         int **S=NULL;
329         S=input_id(S, m);
330         int **T=NULL;
331         T=input_id(T, n);
332         int **At=NULL;
333         At=input_null(At, n, m);
334         int **Tt=NULL;
335         Tt=input_null(Tt, n, n);
336
337         SmithNormalForm(A, S, T, m, n);
338         return;
339 }
340
341 // 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)
342 void SmithNormalForm5mat (int **A, int **S, int **T, int **S_inv, int **T_inv, int
        m, int n) {
343         int **At=NULL;
344         At=input_null(At, n, m);
345         int **Tt=NULL;
346         Tt=input_null(Tt, n, n);
347         int **SinvT=NULL;
348         SinvT=input_null(SinvT, m, m);
349
350         int r=my_min(m, n);
351
352         for (int k=0; k<r; k++) {
353                 Smith_fst5mat(A, S, T, S_inv, T_inv, At, Tt, SinvT, m, n, k);
354         }
355
356         int err=checkDiagZeros(A, r);
357         if (err!=r) {
358                 SinvT=traspMatrix(S_inv, SinvT, m, m);
359                 for (int j=err+1; j<r; j++) {
360                         addRows(A, j, err, -1, n);
361                         addRows(S, j, err, -1, m);
362                         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)
363                 }
364                 S_inv=traspMatrix(SinvT, S_inv, m, m);
365                 for (int k=err; k<r; k++) {
366                         Smith_fst5mat(A, S, T, S_inv, T_inv, At, Tt, SinvT, m, n, k
                                );
367                 }
368         }
369
370         for (int it=0; it<r; it++) {
371                 if (A[it][it]==0) {
372                         break;
373                 }
374                 if (!bool_dk(A, r, it)) {
375                         SinvT=traspMatrix(S_inv, SinvT, m, m);
376                         for (int i=it+1; i<r; i++) {
```

```
377                                 addRows(A, i, it, -1, n);
378                                 addRows(S, i, it, -1, m);
379                                 addRows(SinvT, it, i, 1, m);
380                         }
381                         S_inv=traspMatrix(SinvT, S_inv, m, m);
382                         for (int iter=it; iter!=r; iter++) {
383                                 Smith_fst5mat(A, S, T, S_inv, T_inv, At, Tt, SinvT,
                                        m, n, iter);
384                         }
385                 }
386         }
387         SinvT=traspMatrix(S_inv, SinvT, m, m);
388         switchSigns5mat(A, S, SinvT, m, m, n, r-1);
389         S_inv=traspMatrix(SinvT, S_inv, m, m);
390         return;
391 }
392
393 // come nel caso precedente, questa funzione è analoga a Smith_fst con l'aggiunta
        di S_inv e T_inv
394 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) {
395         while(!nullVector(A, m, k) || !nullVector(traspMatrix(A, At, m, n), n, k))
                {
396                 switchZeros5mat(A, T, T_inv, At, Tt, m, n, k);
397                 SinvT=traspMatrix(S_inv, SinvT, m, m);
398                 while (!nullVector(A, m, k)) {
399                         switchSigns5mat(A, S, SinvT, m, m, n, k);
400                         while (!nullVector(A, m, k)) {
401                                 minUp5mat(A, S, SinvT, m, m, n, k);
402                                 div_rem5mat(A, S, SinvT, m, m, n, k);
403                         }
404                 }
405                 S_inv=traspMatrix(SinvT, S_inv, m, m);
406                 At=traspMatrix(A, At, m, n);
407                 Tt=traspMatrix(T, Tt, n, n);
408                 while (!nullVector(At, n, k)) {
409                         switchSigns5mat(At, Tt, T_inv, n, n, m, k);
410                         while (!nullVector(At, n, k)) {
411                                 minUp5mat(At, Tt, T_inv, n, n, m, k);
412                                 div_rem5mat(At, Tt, T_inv, n, n, m, k);
413                         }
414                 }
415                 A=traspMatrix(At, A, n, m);
416                 T=traspMatrix(Tt, T, n, n);
417         }
418         SinvT=traspMatrix(S_inv, SinvT, m, m);
419         switchSigns5mat(A, S, SinvT, m, m, n, k);
420         S_inv=traspMatrix(SinvT, S_inv, m, m);
421         return;
422 }
423
424 // funzione analoga a switchZeros con l'aggiunta di S_inv e T_inv
425 void switchZeros5mat (int **A, int **T, int **T_inv, int **At, int **Tt, int m, int
        n, int k) {
426         bool var=true;
427         for (int i=k; i<m; i++) {
428                 if (A[i][k]!=0) {
429                         var=false;
430                         break;
431                 }
432         }
433         if (var) {
434                 At=traspMatrix(A, At, m, n);
435                 Tt=traspMatrix(T, Tt, n, n);
436                 for (int j=k; j<(n-1); j++) {
437                         swapRows(At, j, j+1, m);
438                         swapRows(Tt, j, j+1, n);
439                         swapRows(T_inv, j, j+1, n);
440                 }
441                 A=traspMatrix(At, A, n, m);
```

```
442                 T=traspMatrix(Tt, T, n, n);
443         }
444         return;
445 }
446
447 // funzione analoga a switchSigns con l'aggiunta di S_inv e T_inv
448 void switchSigns5mat (int **A, int **mat, int **mat2, int c, int m, int n, int k) {
449         for (int i=k; i<m; i++) {
450                 if (A[i][k]<0) {
451                         for (int j=k; j<n; j++) {
452                                 A[i][j]=-A[i][j];
453                         }
454                         for (int l=0; l<c; l++) {
455                                 mat[i][l]=-mat[i][l];
456                                 mat2[i][l]=-mat2[i][l]; // matrice S_inv o T_inv a
                                        seconda dei casi
457                         }
458                 }
459         }
460         return;
461 }
462
463 // funzione analoga a minUp con l'aggiunta di S_inv e T_inv
464 void minUp5mat (int **A, int **mat, int **mat2, int c, int m, int n, int k) {
465         int min=A[k][k], r_min=k;
466         for(int i=k; i<m; i++) {
467                 if (A[i][k]<min && A[i][k]!=0 || min==0) {
468                         min=A[i][k];
469                         r_min=i;
470                 }
471         }
472         swapRows(A, k, r_min, n);
473         swapRows(mat, k, r_min, c);
474         swapRows(mat2, k, r_min, c); // stesso lavoro anche su S_inv o T_inv a
                seconda dei casi
475
476         return;
477 }
478
479 // funzione analoga a div_rem con l'aggiunta di S_inv e T_inv
480 void div_rem5mat (int **A, int **mat, int **mat2, int c, int m, int n, int k) {
481         int quot=0;
482         for (int i=k+1; i<m; i++) {
483                 quot=A[i][k]/A[k][k];
484                 addRows(A, k, i, quot, n);
485                 addRows(mat, k, i, quot, c);
486                 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)
487         }
488         return;
489 }
490
491 #endif
```

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

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

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 {
```

```
22                  double l_min;
23                  double l_max;
24                  int size;
25                  int** matrix_d;
26                  struct Matrici_Persistenza* next;
27                  struct Matrici_Persistenza* prev;
28          } matrici_persistenza;
29
30          // Mi creo una lista di complessi simpliciali che rappresentano l'
                evoluzione
31          typedef struct Modulo_Persistenza {
32                  double l_min;
33                  double l_max;
34                  int size;
35                  SimplicialComplex* sc;
36                  struct Modulo_Persistenza* next;
37                  struct Modulo_Persistenza* prev;
38          }M_P;
39
40          //matrici_persistenza* create_M_P(void);
41          M_P* create_Modulo_Persistenza(matrici_persistenza*, int);
42          int** matrix_phi_l1_to_l2(Simplex*, int, Simplex*, int, int);
43          int beta_i_j(M_P*, double, double, int, int);
44          int** beta_matrix(int, int, double, M_P*, int, int);
45          int** mu_matrix(int**, int);
46          double** distance_matrix(int**, int);
47          double** distance_matrix_rn(int**, int, int);
48          int** input_point(int*);
49          int** input_point_rn(int*, int);
50          int** zero_one_matrix(double**, int, double);
51          double* find_lambda_value(double**, int);
52          matrici_persistenza* create_matrix_persistenza(double**, int, double*);
53          int compare_double(const void*, const void*);
54          void print1(M_P*, int);
55
56          /* Esempio non da input
57          matrici_persistenza* create_M_P(void) {
58                  matrici_persistenza* mp1 = (matrici_persistenza*)malloc(sizeof(
                        matrici_persistenza));
59                  matrici_persistenza* mp2 = (matrici_persistenza*)malloc(sizeof(
                        matrici_persistenza));
60                  matrici_persistenza* mp3 = (matrici_persistenza*)malloc(sizeof(
                        matrici_persistenza));
61                  matrici_persistenza* mp4 = (matrici_persistenza*)malloc(sizeof(
                        matrici_persistenza));
62                  mp1->l_min = 0.5;
63                  mp1->l_max = 1;
64                  mp2->l_min = 1.5;
65                  mp2->l_max = 10;
66                  mp3->l_min = 10.5;
67                  mp3->l_max = 10.5;
68                  mp4->l_min = 11;
69                  mp4->l_max = 11; // per segnalare la fine
70                  mp1->size = 3;
71                  mp2->size = 3;
72                  mp3->size = 3;
73                  mp4->size = 3;
74                  int** m1 = NULL, ** m2 = NULL, ** m3 = NULL, ** m4 = NULL;
75                  m1 = input_null(m1, 3, 3);
76                  m2 = input_null(m2, 3, 3);
77                  m3 = input_null(m3, 3, 3);
78                  m4 = input_null(m4, 3, 3);
79                  m1[0][0] = 1; m1[0][1] = 0; m1[0][2] = 0;
80                  m1[1][0] = 0; m1[1][1] = 1; m1[1][2] = 0;
81                  m1[2][0] = 0; m1[2][1] = 0; m1[2][2] = 1;
82                  m2[0][0] = 1; m2[0][1] = 1; m2[0][2] = 0;
83                  m2[1][0] = 1; m2[1][1] = 1; m2[1][2] = 0;
84                  m2[2][0] = 0; m2[2][1] = 0; m2[2][2] = 1;
85                  m3[0][0] = 1; m3[0][1] = 1; m3[0][2] = 0;
86                  m3[1][0] = 1; m3[1][1] = 1; m3[1][2] = 1;
```

```
87                  m3[2][0] = 0; m3[2][1] = 1; m3[2][2] = 1;
88                  m4[0][0] = 1; m4[0][1] = 1; m4[0][2] = 1;
89                  m4[1][0] = 1; m4[1][1] = 1; m4[1][2] = 1;
90                  m4[2][0] = 1; m4[2][1] = 1; m4[2][2] = 1;
91                  mp1->matrix_d = m1;
92                  mp2->matrix_d = m2;
93                  mp3->matrix_d = m3;
94                  mp4->matrix_d = m4;
95                  mp1->next = mp2;
96                  mp2->next = mp3;
97                  mp3->next = mp4;
98                  mp4->next = NULL;
99                  mp1->prev = NULL;
100                 mp2->prev = mp1;
101                 mp3->prev = mp2;
102                 mp4->prev = mp3;
103                 return mp1;
104         }*/
105
106         /*Data la successione di matrici creo da quelle i complessi simpliciali
107         salvandomi ogni volta il lambda min e max in cui quel complesso non varia
108         utilizzo la fomra troncata per risparmiare memoria e per alcuni
                accorgimenti detti all'utente*/
109         M_P* create_Modulo_Persistenza(matrici_persistenza* mp, int max) {
110                 M_P* mod_p = NULL, * app = NULL, * my_new = NULL;
111                 int size = mp->size;
112                 mod_p = (M_P*)malloc(sizeof(M_P));
113                 mod_p->l_min = mp->l_min;
114                 mod_p->l_max = mp->l_max;
115                 mod_p->size = size;
116                 mod_p->sc = complex_from_adjacency_matrix_truncated(mp->matrix_d,
                        size, max);
117                 mod_p->prev = NULL;
118                 app = mod_p;
119
120                 while (mp->next) {
121                         mp = mp->next;
122                         size = mp->size;
123                         my_new = (M_P*)malloc(sizeof(M_P));
124                         my_new->l_min = mp->l_min;
125                         my_new->l_max = mp->l_max;
126                         my_new->sc = complex_from_adjacency_matrix_truncated(mp->
                                matrix_d, size, max);
127
128                         my_new->prev = mod_p;
129                         my_new->next = NULL;
130                         mod_p->next = my_new;
131                         mod_p = my_new;
132                 }
133
134                 return app;
135         }
136
137         //Mi costruisco la matrice di cambio di base da s1 a s2 evoluzione dei
                simplessi
138         int** matrix_phi_l1_to_l2(Simplex* s1, int sizeb1, Simplex* s2, int sizeb2,
                 int size) {
139                 int** matrix = NULL;
140                 matrix = input_null(matrix, sizeb2, sizeb1);
141                 Simplex* current = s1;
142                 int col = 0;
143                 //scorro il primo simplesso che è contenuto nel secondo per
                        costruzione e mi salvo il numero
144                 //corrispondente alla posizione e creo la matrice di cambiamento di
                         base
145                 while (current)
146                 {
147                         int* v = current->vertices;
148                         int k = base_number(s2, v, size);
149                         if (k != -1) {
```

```
150                             matrix[k][col] = 1;
151                     }
152                     current = current->next;
153                     col++;
154             }
155
156             return matrix;
157     }
158
159     //calcolo l'elemento ij della matrice beta sempre i > j
160     //n è il grado del complesso i e j l'evoluzione al tempo i e j
161     //h l'omologia cercata
162     int beta_i_j(M_P* mp, double i, double j, int h, int n) {
163             SimplicialComplex* sc1 = NULL, * sc2 = NULL;
164             M_P* app = mp;
165
166             if (i < app->l_min)
167                     return 0;
168             if(h > n)
169                     return 0;
170             //come prima cosa trovo a quale complessi simplicale appartengono
                    le evoluzioni i e j
171             while (app && !sc2) {
172                     if (app->l_min <= i && i <= app->l_max) {
173                             sc1 = app->sc;
174                     }
175
176                     if (app->l_min <= j && j <= app->l_max) {
177                             sc2 = app->sc;
178
179                     }
180                     app = app->next;
181             }
182             //se trovo il secondo anche il primo esiste per costruzione
                    altrimenti torno 0
183             if (!sc2)
184                     return 0;
185
186             //se il primo ha dimensione nulla torno zero perchè vuole dire che
                    non ci sono evoluzioni
187             if (sc1[h].size == 0)
188                     return 0;
189
190             //printf("sc1[h].size = %d\n", sc1[h].size);
191
192             //se tutto è andato bene calcolo il rango dell'omologia tramite phi
193             //per prima cosa trovo la matrice di cambio di base
194             int** phi_i_j = matrix_phi_l1_to_l2(sc1[h].simplices, sc1[h].size,
                    sc2[h].simplices, sc2[h].size, h + 1);
195             //print_matrix(phi_i_j, sc2[h].size, sc1[h].size);
196
197             int** edge1 = NULL;
198             int** kernel_edge1 = NULL;
199             int colonne_base_rango = 0;
200             //mi devo calcolare la matrice di bordo del complesso simpliciale i
201             // tra i simplessi h e h -1
202
203             //quindi se h = 0 la matrice è nulla altrimenti la calcolo e
                    calcolo una base del nucleo
204             if (h == 0) {
205                     //edge1 = input_id(edge1, sc1[h].size);
206                     edge1 = input_null(edge1, sc1[h].size, sc1[h].size);
207                     kernel_edge1 = input_id(kernel_edge1, sc1[h].size);
208                     colonne_base_rango = sc1[h].size;
209             }
210             else {
211                     edge1 = edge_Matrix(sc1, h);
212                     kernel_edge1 = kernel_base(edge1, sc1[h - 1].size, sc1[h].
                            size, &colonne_base_rango);
213             }
```

```
214
215
216
217                 //se il nucleo è nullo bij = 0 perchè non ho simplessi da dove sono
                        partito
218                 if (!kernel_edge1)
219                         return 0;
220
221
222                 //printf("kernel_edge1\n");
223                 //print_matrix(kernel_edge1, sc1[h].size, colonne_base_rango);
224
225                 //altrimenti applico il cambiamento di base al nucleo
226                 int** matrix_j = NULL;
227                 matrix_j = mul_matrix(phi_i_j, sc2[h].size, sc1[h].size,
                        kernel_edge1, sc1[h].size, colonne_base_rango);
228
229                 //non resta che applicare f e calolcare i rispettivi ranghi
230                 int** edge2 = NULL;
231                 int rank2 = 0, row = 0, col = 0;
232
233                 //se h è il massimo allora la matrice di bordo di j è nulla quindi
                        f lo è
234                 // oppure se è nullo lo spazio di partenza
235                 //altrimenti la calcolo
236                 if (h == n)
237                 {
238                         edge2 = input_null(edge2, sc2[h].size, sc2[h].size);
239                         row = sc2[h].size;
240                         col = sc2[h].size;
241                         rank2 = 0;
242                 }
243                 else if (!sc2[h + 1].size || sc2[h + 1].size == 0) {
244
245                         edge2 = input_null(edge2, sc2[h].size, sc2[h].size);
246                         row = sc2[h].size;
247                         col = sc2[h].size;
248                         //rank2 = rank_matrix(edge2, sc2[h].size, sc2[h + 1].size +
                                1);
249                         rank2 = 0;
250
251                 }
252                 else {
253                         edge2 = edge_Matrix(sc2, h + 1);
254                         rank2 = rank_matrix(edge2, sc2[h].size, sc2[h + 1].size);
255                         row = sc2[h].size;
256                         col = sc2[h + 1].size;
257                 }
258
259                 //applico f mettendo le due matrici una di seguito all'altra
260                 int** matrix_f = NULL;
261                 matrix_f = link2matrix_same_row(matrix_j, sc2[h].size,
                        colonne_base_rango, edge2, row, col);
262                 //ne calcolo il rango
263                 int rank1 = rank_matrix(matrix_f, sc2[h].size, col +
                        colonne_base_rango);
264                 //printf("rank1 = %d\n", rank1);
265                 //printf("rank2 = %d\n", rank2);
266                 //il rango dell'omologia è la differenza tra il rango della matrice
                        con f applicata
267                 //e il rango di f
268                 int b = rank1 - rank2;
269
270                 return b;
271
272         }
273
274         //stampo il modulo di persistenza
275         void print1(M_P* a, int n) {
276                 M_P* mp = a;
```

```
277             while (mp) {
278                     printComplex(mp->sc, n);
279                     mp = mp->next;
280             }
281     }
282
283     //calcolo la matrice beta elemento a elemnto ma solo con j >= i
284     // k è il grado del complesso e h è l'omologia richiesta
285     int** beta_matrix(int min, int max, double passo, M_P* mp, int h, int k) {
286             int** matrix = NULL;
287             //numero di passi da calcolare in base al lambda massimo e minimo
288             int n = ((max - min) / passo) + 1;
289             matrix = input_null(matrix, n + 1, n + 1);
290             int i = 0;
291             int j = 0;
292
293             for (i = 0; i <= n; i++) {
294                     for (j = i; j <= n; j++) {
295                             matrix[i][j] = beta_i_j(mp, min + i * passo, min +
                                    j * passo, h, k);
296                     }
297             }
298             return matrix;
299     }
300
301     //mu è una combinazione lineare delle entrate di beta
302     int** mu_matrix(int** beta, int n) {
303             int i = 0, j = 0;
304             int** matrix = NULL;
305             matrix = input_null(matrix, n , n);
306             for (i = 1; i < n; i++)
307                     for (j = i + 1; j < n; j++)
308                             matrix[i][j] = beta[i][j - 1] - beta[i][j] + beta[i
                                    - 1][j] - beta[i - 1][j - 1];
309             return matrix;
310     }
311
312     //calcolo la matrice delle distanze tra i punti nel piano cartesiano
313     double** distance_matrix(int** point, int n) {
314             int i = 0, j = 0;
315             double** matrix = NULL;
316             matrix = (double**)calloc(n, sizeof(double*));
317             for (i = 0; i < n; i++) {
318                     matrix[i] = (double*)calloc(n, sizeof(double));
319             }
320             for (i = 0; i < n; i++) {
321                     for (j = i; j < n; j++) {
322                             matrix[i][j] = sqrt(pow(point[i][0] - point[j][0],
                                    2) + pow(point[i][1] - point[j][1], 2));
323                             matrix[j][i] = matrix[i][j];
324                     }
325             }
326             return matrix;
327     }
328
329     //matrice delle distanze in rk
330     double** distance_matrix_rn(int** point, int n, int k) {
331             int i = 0, j = 0;
332             double** matrix = NULL;
333             double somma = 0;
334             matrix = (double**)calloc(n, sizeof(double*));
335             for (i = 0; i < n; i++) {
336                     matrix[i] = (double*)calloc(n, sizeof(double));
337             }
338             for (i = 0; i < n; i++) {
339                     for (j = i; j < n; j++) {
340                             for (int l = 0; l < k; l++)
341                                     somma += pow(point[l][i] - point[l][j], 2);
342                             matrix[i][j] = sqrt(somma);
343                             matrix[j][i] = matrix[i][j];
```

```
344                             somma = 0;
345                     }
346             }
347             return matrix;
348     }
349
350     //richiede i punti da input
351     int** input_point(int *k) {
352             int** matrix = NULL;
353             int n = 0;
354             printf("Inserisci il numero di punti: n = ");
355             scanf("%d", &n);
356             *k = n;
357             matrix = input_null(matrix, n, 2);
358             int i = 0;
359             for (i = 0; i < n; i++) {
360                     printf("Inserisci le coordinate del punto %d: ", i + 1);
361                     scanf("%d %d", &matrix[i][0], &matrix[i][1]);
362             }
363
364             return matrix;
365     }
366
367     //prendo i punti in rn
368     int** input_point_rn(int* k, int n) {
369             int** matrix = NULL;
370             int l = 0;
371             printf("Inserisci il numero di punti: n = ");
372             scanf("%d", &l);
373             *k = l;
374             matrix = input_null(matrix, l, n);
375             int i = 0;
376             for (i = 0; i < l; i++) {
377                     for (int j = 0; j < n; j++) {
378                             printf("Inserisci le coordinate del punto %d: ", i
                                    + 1);
379                             scanf("%d", &matrix[j][i]);
380                     }
381             }
382
383             return matrix;
384     }
385
386     //calcola le matrici 0-1 partendo dalla matrice delle distanze e un valore
            lambda
387     int** zero_one_matrix(double** matrix, int n, double lambda) {
388             int** m = NULL;
389             m = input_null(m, n, n);
390
391             int i = 0, j = 0;
392             for (i = 0; i < n; i++) {
393                     for (j = i; j < n; j++) {
394                             if (matrix[i][j] < lambda) {
395                                     //printf("1matrix[%d][%d] = %lf\n", i, j,
                                            matrix[i][j]);
396                                     m[i][j] = 1;
397                                     m[j][i] = 1;
398                             }
399                             else
400                             {
401                                     //printf("2matrix[%d][%d] = %lf\n", i, j,
                                            matrix[i][j]);
402                                     m[i][j] = 0;
403                                     m[j][i] = 0;
404                             }
405                     }
406             }
407             return m;
408     }
409
```

```
410         //data la matrice trovo i valori di lambda estremeali
411         //per cui in ogni intervallo trovato non cambia il complesso
412         double* find_lambda_value(double** matrix, int n) {
413                 int size = n * (n - 1) / 2;
414                 double* lambda = (double*)calloc(size, sizeof(double));
415                 int i = 0, j = 0, k = 0;
416
417                 //mi salvo tutti gli elementi sopra la diagonale della matrice
                        delle distanze
418                 for (i = 0; i < n; i++)
419                         for (j = i + 1; j < n; j++) {
420                                 lambda[k] = matrix[i][j];
421                                 k++;
422                         }
423
424
425
426                 double l = 0;
427                 double app = 0;
428                 int iter = 0;
429
430                 //ordino le distanze
431                 qsort(lambda, size, sizeof(double), compare_double);
432
433
434
435                 //creo un vettore con gli estremi degli intervalli andando a
                        leggere le distanze
436                 while (iter < size) {
437                         l += 0.5;
438
439                         if (lambda[iter] == app) {
440                                 lambda[iter] = 0;
441                                 iter++;
442                                 l -= 0.5;
443                                 continue;
444                         }
445                         else if(l > lambda[iter])
446                         {
447                                 app = lambda[iter];
448                                 lambda[iter] = l;
449                                 iter++;
450                         }
451                 }
452
453                 /*printf("lambda\n");
454                 for (int i = 0; i < size; i++)
455                         printf("%lf ", lambda[i]);*/
456                 return lambda;
457         }
458
459         //data la matrice e gli intervallo creo la successione di amtrici di
                adiacenza
460         matrici_persistenza* create_matrix_persistenza(double** matrix, int n,
                double* l) {
461                 matrici_persistenza* mp = NULL, * app = NULL, * my_new = NULL;
462                 int size = (n*(n-1)/2);
463                 //printf("size = %d", size);
464                 double* lambda = find_lambda_value(matrix, n);
465                 /*printf("lambda\n");
466                 for (int i = 0; i < size; i++)
467                         printf("lambda[%d] = %lf\n", i, lambda[i]);*/
468                 //dentro il vettore ci sono zeri che salto
469                 int i = 0;
470                 while (lambda[i] == 0)
471                         i++;
472
473                 //creo quando lambda è non nullo la matrice di adiacenza
474                 //e mi salvo l'intervallo
475                 //per comodità divido il perimo e l'ultimo elemnto della lista
```

```
476                 //poichè hanno valori di minimo e massimo predefiniti
477                 mp = (matrici_persistenza*)malloc(sizeof(matrici_persistenza));
478                 mp->l_min = 0.5;
479                 mp->l_max = lambda[i]-0.5;
480                 //printf("\n\n\n\n Il codice inzia qui");
481                 //printf("l_max = %lf\n", mp->l_max);
482                 //printf("l_min = %lf", mp->l_min);
483                 mp->size = n;
484                 /*printf("matrice n\n");
485                 for (int i = 0; i < n; i++) {
486                         for (int j = 0; j < n; j++) {
487                                 printf("%lf ", matrix[i][j]);
488                         }
489                         printf("\n");
490                 }*/
491                 mp->matrix_d = zero_one_matrix(matrix, n, mp->l_min);
492                 //printf("\n\n\n");
493                 //print_matrix(mp->matrix_d, n, n);
494
495                 mp->prev = NULL;
496                 mp->next = NULL;
497                 app = mp;
498                 i++;
499
500                 while (i < size)
501                 {
502                         my_new = (matrici_persistenza*)malloc(sizeof(
                            matrici_persistenza));
503                         my_new->l_min = mp->l_max + 0.5;
504                         while (lambda[i] == 0 && i < size)
505                         {
506                                         i++;
507                         }
508                         if (i < size) {
509                                 my_new->l_max = lambda[i] - 0.5;
510                                 //printf("l_max = %lf\n", my_new->l_max);
511                                 my_new->size = n;
512                                 my_new->matrix_d = zero_one_matrix(matrix, n,
                                        my_new->l_max);
513                                 //printf("\n\n\n");
514                                 //print_matrix(my_new->matrix_d, n, n);
515                                 my_new->prev = mp;
516                                 my_new->next = NULL;
517                                 mp->next = my_new;
518                                 mp = my_new;
519                                 i++;
520                                 //printf("l_min = %lf\n", my_new->l_min);
521                                 //printf("l_max = %lf\n", my_new->l_max);
522                         }
523
524                 }
525
526                 my_new = (matrici_persistenza*)malloc(sizeof(matrici_persistenza));
527                 my_new->l_min = mp->l_max + 0.5;
528
529                 my_new->l_max = my_new->l_min;
530                 my_new->size = n;
531                 //printf("l_max = %lf\n", my_new->l_min);
532                 my_new->matrix_d = zero_one_matrix(matrix, n, my_new->l_max);
533                 //printf("Matrice n");
534                 //print_matrix(my_new->matrix_d, n, n);
535                 my_new->prev = mp;
536                 my_new->next = NULL;
537                 mp->next = my_new;
538                 mp = my_new;
539                 l[0] = mp->l_max;
540
541                 return app;
542         }
543
```

```
544         //funzione per comparare due double usata successivamente
545         int compare_double(const void* a, const void* b) {
546                 double da = *(const double*)a;
547                 double db = *(const double*)b;
548 
549                 if (da < db) return -1;
550                 if (da > db) return 1;
551                 return 0;
552         }
553 
554 #endif
```

Listing 9: Grafico Barcode

```
1  import sys
2  import json
3  import matplotlib.pyplot as plt
4  
5  
6  """print("Argomenti ricevuti:", sys.argv)
7  """
8  def plot_barcode(matrix, lambda_min, h):
9      """
10     Data una matrice quadrata (lista di liste) che rappresenta
11     il modulo di persistenza, estrae gli intervalli e disegna il barcode.
12 
13     Regole:
14       - Per ogni riga non nulla (con almeno un valore non zero):
15           * Si parte dalla riga (birth = numero della riga, 1-indexed).
16           * Si individuano gli elementi non nulli, a partire dall'elemento più a
                 destra
17             e poi procedendo verso sinistra.
18           * Per ciascun elemento non nullo, lâ€™intervallo è [birth, death] dove:
19               - birth = numero della riga (1-indexed);
20               - death = numero della colonna (1-indexed).
21       - Ogni intervallo viene disegnato come segmento orizzontale e posizionato
             sulla "prima quota libera"
22         (cioè, sul livello y più basso in cui non si sovrappone ad un altro
               intervallo).
23     """
24     n = len(matrix)  # dimensione della matrice
25     intervals = []   # lista di intervalli: ciascuno è una tupla (birth, death)
26 
27     # Scorriamo le righe (i indici partono da 0; aggiungiamo 1 per ottenere la 1-
           indexazione)
28     for i, row in enumerate(matrix):
29         if any(val != 0 for val in row):
30             # Troviamo gli indici degli elementi non nulli e li ordiniamo
                   decrescentemente (da destra a sinistra)
31             nonzero_indices = [j for j, val in enumerate(row) if val != 0]
32             nonzero_indices.sort(reverse=True)
33 
34             birth = lambda_min + i*h   # riga corrente in 1-indexing
35             for j in nonzero_indices:
36                 death = lambda_min + j*h  # colonna in 1-indexing
37                 num_segments = row[j]
38                 # Aggiungiamo tante istanze quanto il valore in cella
39                 for _ in range(num_segments):
40                     intervals.append((birth, death))
41 
42 
43     # Ordiniamo gli intervalli per birth (e per death in ordine decrescente se la
           riga è la stessa)
44     intervals.sort(key=lambda x: (x[0], -x[1]))
```

```
45
46      # Assegniamo la quota (livello) ad ogni intervallo
47      levels = []  # levels[l] contiene il valore di death dell'ultimo intervallo
            assegnato a quel livello
48      assigned_intervals = []  # lista di tuple (birth, death, livello)
49
50      for birth, death in intervals:
51          assigned_level = None
52          # Cerchiamo il livello l più basso per cui l'intervallo corrente non si
                sovrappone
53          for l, last_death in enumerate(levels):
54              if last_death < birth:
55                  assigned_level = l
56                  break
57          if assigned_level is None:
58              assigned_level = len(levels)
59              levels.append(0)
60          levels[assigned_level] = death
61          assigned_intervals.append((birth, death, assigned_level + 1))  # usiamo 1-
                indexazione per il livello
62
63      # Disegniamo il barcode con matplotlib
64      fig, ax = plt.subplots()
65      for birth, death, level in assigned_intervals:
66          ax.hlines(level, birth, death, colors='blue', lw=2)
67          ax.plot([birth, death], [level, level], 'bo')
68
69      ax.set_xlabel("x (indice)")
70      ax.set_ylabel("Quota (livello)")
71      ax.set_title("Barcode del modulo di persistenza")
72      # Impostiamo i limiti dell'asse x:
73      # da un po' sotto lambda_min fino a lambda_min + n*h (o anche oltre, se
            necessario)
74      ax.set_xlim(lambda_min, lambda_min + (n-1) * h)
75
76      # Impostiamo i tick delle ascisse: ogni riga corrisponde a un tick
77      xticks = [lambda_min + i * h for i in range(n)]
78      ax.set_xticks(xticks)
79      ax.set_xticklabels([f"{x:.2f}" for x in xticks])
80      max_level = max((lev for (_, _, lev) in assigned_intervals), default=1)
81      ax.set_ylim(0, max_level + 1)
82
83      plt.grid(axis='x', linestyle='--', alpha=0.6)
84      plt.show()
85
86  def main():
87      # Se viene passato un argomento, lo interpretiamo come una stringa JSON
            contenente la matrice
88      if len(sys.argv) > 1:
89          matrix_str = sys.argv[1]
90          try:
91              matrix = json.loads(matrix_str)
92          except json.JSONDecodeError as e:
93              print("Errore nella decodifica della matrice:", e)
94              sys.exit(1)
95          lambda_min = float(sys.argv[2])  # Valore minimo della scala delle ascisse
96          h = float(sys.argv[3])  # Passo della scala delle ascisse
97      else:
98          # Matrice di default (se non viene passato alcun argomento)
99          matrix = [
100             [0, 0, 0, 0, 0],
101             [0, 0, 0, 0, 0],
102             [0, 0, 0, 1, 1],
103             [0, 0, 0, 0, 0],
104             [0, 0, 1, 1, 0]
105         ]
106     plot_barcode(matrix, lambda_min, h)
107
108 if __name__ == '__main__':
109     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 $(d) := < \{d\} > = \{ad\ | a \in A\}$ 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;
}

```

```
28 // massimo comun divisore tra n e m, salvato in a, con algoritmo binario. Il
       risultato è sempre non negativo.
29 void mcd_binario (mpz_t a, mpz_t n, mpz_t m) {
30     // casi base: uno dei due numeri è 0
31     if (mpz_cmp_si(n,0)==0) {
32         mpz_set(a,m);
33         mpz_abs(a,a);
34         return;
35     }
36     else if (mpz_cmp_si(m,0)==0) {
37         mpz_set(a,n);
38         mpz_abs(a,a);
39         return;
40     }
41
42     // creo nuove variabili per non modificare n e m, e le rendo positive per avere
           mcd>=0
43     mpz_t x,y;
44     mpz_inits(x,y,NULL);
45     mpz_abs(x,n); mpz_abs(y,m);
46
47  /* ottimizzazione che ho trovato io facendo qualche test: se i due numeri hanno
       ordini di grandezza molto diversi l'algoritmo diventa
48     inefficiente, quindi eseguo una sola passata dell'algoritmo euclideo facendo
           una prima divisione con resto. In questo modo i due
49     numeri avranno ordine di grandezza simile al più piccolo, e l'algoritmo binario
           si velocizza molto. */
50     if (mpz_cmp(x,y)>0) {
51         mpz_fdiv_r(x,x,y);
52         if (mpz_cmp_si(x,0)==0) {
53             mpz_set(a,y);
54             mpz_clears(x,y,NULL);
55             return;
56         }
57     }
58     else {
59         mpz_fdiv_r(y,y,x);
60         if (mpz_cmp_si(y,0)==0) {
61             mpz_set(a,x);
62             mpz_clears(x,y,NULL);
63             return;
64         }
65     }
66
67     // calcolo (in k) la più alta potenza di 2 che divide i due numeri
68     unsigned long int i,j,k;
69     i = mpz_scan1(x,0);
70     j = mpz_scan1(y,0);
71     if (i<j) k=i;
72     else k=j;
73
74     // rendo dispari i due numeri, avendo già salvato la più alta potenza di 2 che
           divide entrambi
75     mpz_fdiv_q_2exp(x,x,i);
76     mpz_fdiv_q_2exp(y,y,j);
77
78     // ciclo lavorando sempre su numeri dispari, ed esco quando uno dei due è 0
79     while (mpz_cmp_si(x,0)>0) {
80         if (mpz_cmp(x,y)<0) mpz_swap(x,y); // in questo modo x>=y
81         mpz_sub(x,x,y);  // eseguo la sottrazione x=x-y; ora x è pari
82         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
83         i = mpz_scan1(x,0);
84         mpz_fdiv_q_2exp(x,x,i);  // tolgo tutte le potenze di 2 che dividono x: cos
               ì torna dispari
85     }
86
87     // quando esco dal ciclo in y ci sarà mcd dei numeri resi dispari; dopo
           rimoltiplico per la giusta potenza di 2 calcolata prima
```

```
88      mpz_set(a,y);
89      mpz_mul_2exp(a,a,k);
90
91      mpz_clears(x,y,NULL);
92      return;
93  }
94
95  // massimo comun divisore degli n>0 interi dell'array "integers", salvato in a, con
        algoritmo di Euclide esteso. Il risultato è sempre non negativo.
96  void mcd_euclide_array (mpz_t a, mpz_t* integers, int n) {
97      // caso banale: n=1
98      if (n==1) {
99          mpz_set(a,integers[0]);
100         mpz_abs(a,a);
101         return;
102     }
103     // creo nuove variabili per non modificare gli interi dati, e le rendo positive
            per avere mcd>=0
104     int i;
105     mpz_t* nums = malloc(n*sizeof(mpz_t));
106     for (i=0; i<n; i++) {
107         mpz_init_set(nums[i],integers[i]);
108         mpz_abs(nums[i],nums[i]);
109     }
110
111     int k=n-1; i=0;
112     // sposto gli zeri in fondo
113     while (i<k) {
114         while (mpz_cmp_si(nums[i],0)>0 && i<n) i++;
115         if (i>=n) break;
116         while (mpz_cmp_si(nums[k],0)==0 && k>=0) k--;
117         if (k<0) break;
118         if (i>=k) break;
119         // ho individuato in posizione i un elemento ==0 e in posizione k un
                elemento >0: li scambio
120         mpz_swap(nums[i],nums[k]);
121         i++; k--;
122     }
123     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)
124     for (i=0; i<n; i++) {
125         if (mpz_cmp_si(nums[i],0)==0) {
126             k=i;
127             break;
128         }
129     }
130     // caso vettore tutto nullo: restituisco mcd=0
131     if (k==0) {
132         mpz_set_si(a,0);
133         for (i=0; i<n; i++) mpz_clear(nums[i]);
134         return;
135     }
136
137     // ciclo finché ho più di un elemento non nullo
138     while (k>1) {
139         // trovo il minimo elemento non nullo e lo porto in prima posizione
140         int index_min=0;
141         for (i=1; i<k; i++) {
142             if (mpz_cmp(nums[index_min],nums[i])>0) index_min=i;
143         }
144         mpz_swap(nums[index_min],nums[0]);
145
146         // divido per il minimo, calcolo i resti e sposto eventuali zeri ottenuti
147         for (i=1; i<k; i++) {
148             mpz_fdiv_r(nums[i],nums[i],nums[0]); // nums[i] diventa nums[i] mod
                    nums[0], dove nums[0] è il minimo
149
150             // se dopo la divisione ho ottenuto resto 0, porto il numero in fondo
151             if (mpz_cmp_si(nums[i],0)==0) {
152                 mpz_swap(nums[i],nums[k-1]);
```

```
153                 k--; i--; // aggiorno gli indici
154             }
155         }
156     }
157 
158     // quando esco dal ciclo ho k=1 ossia solo il primo elemento è non nullo: ho
        trovato mcd.
159     mpz_set(a,nums[0]);
160     for (i=0; i<n; i++) mpz_clear(nums[i]);
161     return;
162 }
163 
164 // massimo comun divisore degli n>0 interi dell'array "integers", salvato in a, con
     algoritmo binario esteso. Il risultato è sempre non negativo.
165 void mcd_binario_array (mpz_t a, mpz_t* integers, int n) {
166     // caso banale: n=1
167     if (n==1) {
168         mpz_set(a,integers[0]);
169         mpz_abs(a,a);
170         return;
171     }
172     // creo nuove variabili per non modificare gli interi dati, e le rendo positive
         per avere mcd>=0
173     int i;
174     mpz_t* nums = malloc(n*sizeof(mpz_t));
175     for (i=0; i<n; i++) {
176         mpz_init_set(nums[i],integers[i]);
177         mpz_abs(nums[i],nums[i]);
178     }
179 
180     int k=n-1; i=0;
181     // sposto gli zeri in fondo
182     while (i<k) {
183         while (mpz_cmp_si(nums[i],0)>0 && i<n) i++;
184         if (i>=n) break;
185         while (mpz_cmp_si(nums[k],0)==0 && k>=0) k--;
186         if (k<0) break;
187         if (i>=k) break;
188         // ho individuato in posizione i un elemento ==0 e in posizione k un
            elemento >0: li scambio
189         mpz_swap(nums[i],nums[k]);
190         i++; k--;
191     }
192     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)
193     for (i=0; i<n; i++) {
194         if (mpz_cmp_si(nums[i],0)==0) {
195             k=i;
196             break;
197         }
198     }
199     // caso vettore tutto nullo: restituisco mcd=0
200     if (k==0) {
201         mpz_set_si(a,0);
202         for (i=0; i<n; i++) mpz_clear(nums[i]);
203         return;
204     }
205 
206  /* ottimizzazione che ho trovato io facendo qualche test: se i numeri hanno ordini
       di grandezza molto diversi l'algoritmo diventa
207     inefficiente, quindi eseguo una sola passata dell'algoritmo euclideo facendo
        una prima divisione con resto. In questo modo i
208     numeri avranno ordine di grandezza simile al più piccolo, e l'algoritmo binario
         si velocizza molto. */
209     int index_min=0;
210     for (i=1; i<k; i++) {
211         if (mpz_cmp(nums[index_min],nums[i])>0) index_min=i;
212     }
213     mpz_swap(nums[index_min],nums[0]);
214     // divido per il minimo, calcolo i resti e sposto eventuali zeri ottenuti
```

```
215     for (i=1; i<k; i++) {
216         mpz_fdiv_r(nums[i],nums[i],nums[0]); // nums[i] diventa nums[i] mod nums
                [0], dove nums[0] è il minimo
217         // se dopo la divisione ho ottenuto resto 0, porto il numero in fondo
218         if (mpz_cmp_si(nums[i],0)==0) {
219             mpz_swap(nums[i],nums[k-1]);
220             k--; i--; // aggiorno gli indici
221         }
222     }
223 
224     // inizio (finalmente) con la parte di algoritmo binario: salvo la più alta
            potenza di 2 che divide tutti i numeri
225     unsigned long int j,exp;
226     exp=mpz_scan1(nums[0],0);
227     mpz_fdiv_q_2exp(nums[0],nums[0],exp);
228     for (i=1; i<k; i++) {
229         j=mpz_scan1(nums[i],0);
230         mpz_fdiv_q_2exp(nums[i],nums[i],j);
231         if (j<exp) exp=j;
232     }
233 
234     // ciclo finché ho più di un elemento non nullo; saranno sempre dispari
235     while (k>1) {
236         // trovo il minimo elemento non nullo e lo porto in prima posizione
237         int index_min=0;
238         for (i=1; i<k; i++) {
239             if (mpz_cmp(nums[index_min],nums[i])>0) index_min=i;
240         }
241         mpz_swap(nums[index_min],nums[0]);
242 
243         // sottraggo il minimo, poi se ho ottenuto 0 lo sposto in fondo, altrimenti
                tolgo le potenze di 2
244         for (i=1; i<k; i++) {
245             mpz_sub(nums[i],nums[i],nums[0]);
246 
247             if (mpz_cmp_si(nums[i],0)==0) {
248                 mpz_swap(nums[i],nums[k-1]);
249                 k--; i--;
250             }
251             else {
252                 j=mpz_scan1(nums[i],0);
253                 mpz_fdiv_q_2exp(nums[i],nums[i],j);
254             }
255         }
256     }
257 
258     // 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
259     mpz_set(a,nums[0]);
260     mpz_mul_2exp(a,a,exp);
261     for (i=0; i<n; i++) mpz_clear(nums[i]);
262     return;
263 }
264 
265 // massimo comun divisore degli n>0 interi dell'array "integers", salvato in a,
        ottenuto concatenando l'algoritmo euclideo a coppie
266 void mcd_euclide_concat (mpz_t a, mpz_t* integers, int n) {
267     // caso banale: n=1
268     if (n==1) {
269         mpz_set(a,integers[0]);
270         mpz_abs(a,a);
271         return;
272     }
273 
274     mcd_euclide(a,integers[0],integers[1]);
275     for (int i=2; i<n; i++) {
276         mcd_euclide(a,a,integers[i]);
277     }
278 
279     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
```

```
34                  mpz_neg(q,q);
35                  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
36                  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
37                  mpz_set(M[0][0],temp1); mpz_set(M[0][1],temp2); // calcolata riga 1
38      }
39      mpz_set(d,a);  // calcolato mcd
40          mpz_set(x,M[0][0]); mpz_set(y,M[0][1]);
41          // aggiusto i coefficienti per eventuali cambi di segno
42          if (mpz_cmp_si(n,0)<0) mpz_neg(x,x);
43          if (mpz_cmp_si(m,0)<0) mpz_neg(y,y);
44
45          for (i=0; i<2; i++) {
46                  for (j=0; j<2; j++) mpz_clear(M[i][j]);
47          }
48          mpz_clears(a,b,q,temp1,temp2,NULL);
49          return;
50  }
51
52  // 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]
53  void bezout_array (mpz_t d, mpz_t* coeff, mpz_t* integers, int n) {
54          // caso banale: n=1
55      if (n==1) {
56          mpz_set(d,integers[0]);
57                  if (mpz_cmp_si(d,0)<0) {
58                          mpz_neg(d,d);
59                          mpz_set_si(coeff[0],-1);
60                  }
61                  else mpz_set_si(coeff[0],1);
62          return;
63      }
64
65          // creo nuove variabili per non modificare gli interi dati (positive per
                avere mcd>=0), e alloco la matrice necessaria per calcolare i
                coefficienti
66      int i,j;
67      mpz_t* nums = malloc(n*sizeof(mpz_t));
68      for (i=0; i<n; i++) {
69          mpz_init_set(nums[i],integers[i]);
70          mpz_abs(nums[i],nums[i]);
71      }
72          mpz_t** M=malloc(n*sizeof(mpz_t*));
73          for (i=0; i<n; i++) M[i]=malloc(n*sizeof(mpz_t));
74          init_id_matrix(M,n);
75
76          // algoritmo di euclide:
77
78          int k=n-1; i=0;
79      // sposto gli zeri in fondo
80      while (i<k) {
81          while (mpz_cmp_si(nums[i],0)>0 && i<n) i++;
82          if (i>=n) break;
83          while (mpz_cmp_si(nums[k],0)==0 && k>=0) k--;
84          if (k<0) break;
85          if (i>=k) break;
86          // ho individuato in posizione i un elemento ==0 e in posizione k un
                elemento >0: li scambio, e tengo traccia nella matrice M
87          mpz_swap(nums[i],nums[k]);
88                  swap_rows(M,n,i,k);
89          i++; k--;
90      }
91          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)
92      for (i=0; i<n; i++) {
93          if (mpz_cmp_si(nums[i],0)==0) {
94              k=i;
95              break;
96          }
```

```
97      }
98      // caso vettore tutto nullo: restituisco d=0, coeff = array nullo.
99      if (k==0) {
100         mpz_set_si(d,0);
101         for (i=0; i<n; i++) mpz_set_si(coeff[i],0);
102                 for (i=0; i<n; i++) mpz_clear(nums[i]);
103                 for (i=0; i<n; i++) for (j=0; j<n; j++) mpz_clear(M[i][j]);
104         return;
105     }
106
107         mpz_t q; mpz_init(q);
108         // ciclo finché ho più di un elemento non nullo
109     while (k>1) {
110         // trovo il minimo elemento non nullo e lo porto in prima posizione,
                tenendo traccia in M
111         int index_min=0;
112         for (i=1; i<k; i++) {
113             if (mpz_cmp(nums[index_min],nums[i])>0) index_min=i;
114         }
115         mpz_swap(nums[index_min],nums[0]);
116                 swap_rows(M,n,index_min,0);
117
118         // divido per il minimo, calcolo i resti e sposto eventuali zeri ottenuti
119         for (i=1; i<k; i++) {
120             mpz_fdiv_qr(q,nums[i],nums[i],nums[0]); // nums[i] diventa nums[i] mod
                    nums[0], dove nums[0] è il minimo
121                     mpz_neg(q,q);
122                     add_multiple_row(M,n,q,0,i); // alla riga i tolgo q volte
                            la prima riga (con indice 0)
123
124             // se dopo la divisione ho ottenuto resto 0, porto il numero in fondo
125             if (mpz_cmp_si(nums[i],0)==0) {
126                 mpz_swap(nums[i],nums[k-1]);
127                                 swap_rows(M,n,i,k-1);
128                 k--; i--; // aggiorno gli indici
129             }
130         }
131     }
132
133         // 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
134     mpz_set(d,nums[0]);
135         for (i=0; i<n; i++) {
136                 mpz_set(coeff[i],M[0][i]);
137                 if (mpz_cmp_si(integers[i],0)<0) mpz_neg(coeff[i],coeff[i]); //
                        aggiusto i segni poiché all'inizio avevo reso tutto positivo
138         }
139
140         mpz_clear(q);
141         for (i=0; i<n; i++) mpz_clear(nums[i]);
142         for (i=0; i<n; i++) for (j=0; j<n; j++) mpz_clear(M[i][j]);
143         return;
144 }
145
146 // Calcola in x l'inverso di a modulo n: se esiste restituisce 1, altrimenti 0.
        Scelgo x>=0, e suppongo n>0.
147 int inv_mod (mpz_t x, mpz_t a, mpz_t n) {
148         mpz_t d,y;
149         mpz_inits(d,y,NULL);
150
151         bezout(d,x,y,a,n);
152         mpz_fdiv_r(x,x,n); // rendo x tale che 0<x<n
153         if (mpz_cmp_si(d,1)==0) { // caso mcd(a,n)=1: l'inverso esiste ed è x
154                 mpz_clears(d,y,NULL);
155                 return 1;
156         }
157         // caso mcd(a,n)!=1: l'inverso non esiste
158         mpz_clears(d,y,NULL);
159         return 0;
160 }
```

```
161
162 // Come la funzione precedente, ma restituisco anche d=mcd(a,n)
163 int inv_mod_mcd (mpz_t d, mpz_t x, mpz_t a, mpz_t n) {
164         mpz_t y;
165         mpz_init(y);
166
167         bezout(d,x,y,a,n);
168         mpz_fdiv_r(x,x,n); // rendo x tale che 0<x<n
169         if (mpz_cmp_si(d,1)==0) { // caso mcd(a,n)=1: l'inverso esiste ed è x
170                 mpz_clear(y);
171                 return 1;
172         }
173         // caso mcd(a,n)!=1: l'inverso non esiste
174         mpz_clear(y);
175         return 0;
176 }
177
178 // calcola in x la base b elevata alla potenza exp>=0 modulo n, con esponenziazione
      binaria
179 void exp_mod (mpz_t x, mpz_t b, mpz_t exp, mpz_t n) {
180         int bit;
181         mpz_t squares,e;
182         mpz_init_set(squares,b);
183         mpz_init_set(e,exp);
184         mpz_set_si(x,1);
185
186         while (mpz_cmp_si(e,0)>0) {
187                 bit=mpz_tstbit(e,0); // controllo l'ultima cifra della
                      rappresentazione binaria di e
188                 if (bit==1) { // se bit è 1 moltiplico x per il quadrato (mod n),
                      altrimenti lascio così
189                         mpz_mul(x,x,squares);
190                         mpz_fdiv_r(x,x,n);
191                 }
192                 mpz_mul(squares,squares,squares); // aggiorno il quadrato (mod n)
193                 mpz_fdiv_r(squares,squares,n);
194                 mpz_fdiv_q_2exp(e,e,1); // shifto di 1 i bit di e
195         }
196
197         mpz_clears(squares,e,NULL);
198         return;
199 }
200
201 // come exp_mod sopra, ma l'esponente è di tipo unsigned int
202 void exp_mod_ui (mpz_t x, mpz_t b, unsigned int exp, mpz_t n) {
203         mpz_t e;
204         mpz_init_set_ui(e,exp);
205         exp_mod(x,b,e,n);
206         mpz_clear(e);
207
208         return;
209 }
210
211 // inizializza M matrice identità n*n
212 void init_id_matrix (mpz_t** M, int n) {
213         int i,j;
214         for (i=0; i<n; i++) {
215                 for (j=0; j<n; j++) {
216                         mpz_init(M[i][j]);
217                 }
218         }
219         for (i=0; i<n; i++) mpz_set_si(M[i][i],1);
220
221         return;
222 }
223
224 // scambia le righe i e k della matrice M (n*n)
225 void swap_rows (mpz_t** M, int n, int i, int k) {
226         int j;
227         for (j=0; j<n; j++) mpz_swap(M[i][j],M[k][j]);
```

```
228
229         return;
230 }
231
232 // nella matrice M n*n, aggiungo alla riga k la riga i moltiplicata per mult
233 void add_multiple_row (mpz_t** M, int n, mpz_t mult, int i, int k) {
234         int j;
235         mpz_t temp;
236         mpz_init(temp);
237         for (j=0; j<n; j++) {
238                 mpz_mul(temp,M[i][j],mult);
239                 mpz_add(M[k][j],M[k][j],temp);
240         }
241
242         mpz_clear(temp);
243         return;
244 }
```

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

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <stdbool.h>
4  #include "..\Include\Smith.h"
5  #include "..\Include\Bezout_old.h"
6
7  int** input_sys (int**, int);
8  void print_sys (int**, int);
9  int* input_vec_zeros (int*, int);
10 int* input_vec_one (int*, int);
11 void solve_sys (int**, int*, int*, int*, int*, int);
12 int adjust_sol (int, int);
13
14 int main() {
15         int m;
16         printf("Numero di equazioni nel sistema: ");
17         scanf("%d", &m);
18         printf("Dati del sistema (ad x=a mod b corrisponde l'input da tastiera 'a b
               '):\n");
19         int **sys=NULL;
20         sys=input_sys(sys, m);
21         print_sys(sys, m);
22
23         // vettori necessari per la risoluzione del sistema
24         int *A=NULL; // sono i moduli delle singole equazioni
25         A=input_vec_zeros(A, m);
26         int *B=NULL; // B[i] è il prodotto di A[j] con j!=i
27         B=input_vec_one(B, m);
28         int *Alpha=NULL; // sono le congruenze delle singole equazioni
```

```
29          Alpha=input_vec_zeros(Alpha, m);
30          int *Gamma=NULL; // conterrà la prima riga di S dopo aver applicato Smith a
                B
31          Gamma=input_vec_zeros(Gamma, m);
32          solve_sys(sys, A, B, Alpha, Gamma, m);
33
34          // troviamo la soluzione x
35          int x=0;
36          int mod=1;
37          for (int i=0; i<m; i++) {
38                  x=x+Gamma[i]*B[i]*Alpha[i];
39                  mod=mod*A[i];
40          }
41          // aggiustiamo la soluzione x rendendola della forma x=a mod b con 0<=a<b
42          x=adjust_sol(x, mod);
43          printf ("Soluzione: x=%d mod %d\n", x, mod);
44
45          return 0;
46  }
47
48  // inserire la matrice dei dati del sistema (matrice m*2)
49  int** input_sys (int** mat, int m) {
50          mat=calloc(m, sizeof(int*));
51          for (int i=0; i<m; i++) {
52                  mat[i]=calloc(2, sizeof(int));
53          }
54          for (int i=0; i<m; i++) {
55                  for (int j=0; j<2; j++) {
56                          scanf("%d", &mat[i][j]);
57                  }
58          }
59          return mat;
60  }
61
62  // stampare il sistema
63  void print_sys (int** sys, int m) {
64          printf("\nSistema: ");
65          for (int i=0; i<m; i++) {
66                  printf("x=%d mod %d\n", sys[i][0], sys[i][1]);
67                  if (i!=(m-1)) {
68                          printf("\t ");
69                  }
70          }
71          printf("\n");
72          return;
73  }
74
75  // inizializzare un vettore nullo di m componenti
76  int* input_vec_zeros (int *vec, int m) {
77          vec=calloc(m, sizeof(int));
78          return vec;
79  }
80
81  // inizializzare un vettore di 1 di m componenti
82  int* input_vec_one (int* vec, int m) {
83          vec=calloc(m, sizeof(int));
84          for (int i=0; i<m; i++) {
85                  vec[i]=1;
86          }
87          return vec;
88  }
89
90  // creare i vettori che serviranno poi nella risoluzione del sistema
91  void solve_sys (int **sys, int *A, int *B, int *Alpha, int *Gamma, int m) {
92          for (int i=0; i<m; i++) { // prese congruenze del tipo x=a mod alpha
93                  A[i]=sys[i][1]; // A è il vettore delle a
94                  for (int j=0; j<m; j++) {
95                          if (j!=i) {
96                                  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);
```

```
8  long int** inputMatPot (long int**, int*, int, long int, long int);
9  long int potAmod (long int, long int);
10 long int congruenza_a (long int, long int, long int);
11 long int adjust_x (long int, long int);
12 long int change_a (long int, long int, long int);
13 bool fermat (long int, int);
14 long int MCD(long int, long int);
15
16 long int** matNull (long int **mat, int m, int n) {
17         mat=calloc(m, sizeof(long int*));
18         for (int i=0; i<m; i++) {
19                 mat[i]=calloc(n, sizeof(long int));
20         }
21         return mat;
22 }
23
24 int* trasfBin (int *vecBin, long int n, int l) {
25         vecBin=calloc(l, sizeof(int));
26         for (int i=l-1; i>=0; i--) {
27                 vecBin[i]=n%2;
28                 n=n/2;
29         }
30         return vecBin;
31 }
32
33 void printVec (int *vecBin, int l) {
34         for (int i=0; i<l; i++) {
35                 printf("%d", vecBin[i]);
36         }
37         printf("\n");
38         return;
39 }
40
41 long int** inputMatPot (long int **matPot, int *vecBin, int l, long int a, long int
       n) {
42         matPot=matNull(matPot, l, 2);
43         int j=l-1;
44         for (int i=0; i<l; i++) {
45                 matPot[i][0]=vecBin[j]; //nella prima colonna scrivimao il numero
                      in base binaria (al contrario)
46                 j=j-1;
47                 if (i==0) { // nella seconda colonna mettiamo le potenze di a mod n
                       (a, a^2, (a^2)^2 ...)
48                         matPot[i][1]=a;
49                 } else {
50                         matPot[i][1]=potAmod(matPot[i-1][1], n);
51                 }
52         }
53         return matPot;
54 }
55
56 // calcola l'elevazione al quadrato di k mod mod
57 long int potAmod (long int k, long int mod) {
58         long int x=k*k;
59         x=adjust_x(x,mod);
60         return x;
61 }
62
63 // calcola a^(n-1) mod n
64 long int congruenza_a (long int a, long int m, long int n) {
65         int lBin=floor(log2(m))+1; // calcolo la lunghezza di n-1 (ossia m) in
              binario per poi inizializzare il vettore
66         int *vecBin=NULL;
67         vecBin=trasfBin(vecBin, m, lBin); //m sarà n-1, quindi trasformo n-1 in
              binario
68         long int **matPot=NULL;
69         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)
```

```
70          long int c=1; // inizializziamo c=a^(n-1) mod n
71          for (int i=0; i<lBin; i++) { // modifichiamo c a dovere
72                  if (matPot[i][0]==1) { // se la cifra in binario è 1 moltiplico c
                        per la potenza di a corrispondente, sennò non modifico c
73                          c=c*matPot[i][1];
74                  }
75                  c=adjust_x(c, n); // riporto c congruo modulo n
76          }
77          return c;
78  }
79
80  // prendo la classe di congruenza mod mod di x
81  long int adjust_x (long int x, long int mod) {
82          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)
83                  long int temp=0;
84                  temp=x/mod;
85                  x=x-temp*mod;
86          }
87          long int ceilModMezzi=(mod/2)+1; // perchè ceil lavora su double
88          if (x>ceilModMezzi) { // prendiamo la classe di equivalente di modulo
                minore
89                  x=x-mod;
90          } else if (x<0 && (-x)>=ceilModMezzi) {
91                  x=mod+x;
92          }
93          return x;
94  }
95
96  // modifico in maniera casuale la base a
97  long int change_a (long int a, long int max, long int min) {
98          srand(time(NULL));
99          a=rand()%(max-min+1)+min; // prendiamo una base a scelta casualmente nell'
                intervallo [2, n-2]
100         return a;
101 }
102
103 // studiamo la primalità dei numeri dispari
104 bool fermat (long int n, int it_max) {
105         long int a=2, max=n-2, min=2;
106         for (int it=1; it<=it_max; it++) {
107                 long int d=MCD(a, n); // calcolo il massimo comune divisore tra a e
                        n
108                 if (d!=1) { // se è diverso da 1 n non è primo e d è un suo
                        divisore
109                         printf("%ld non è un numero primo e %ld è un suo divisore\n
                                ", n, d);
110                         return false;
111                 } 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
112                         long int c=congruenza_a(a, n-1, n);
113                         if(c!=1) {
114                                 printf("%ld non è un numero primo\n", n);
115                                 return false;
116                         } else {
117                                 a=change_a(a, max, min);
118                         }
119                 }
120         }
121         return true;
122 }
123
124 long int MCD(long int a, long int b) {
125         long int temp=0;
126         while (b!=0) {
127                 temp=b;
128                 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
```

```
54                  if (mpz_cmp_si(d, 1)!=0) { // se d!=1, n non è primo e d è un suo
                        divisore
55                      gmp_printf("%Zd non è un numero primo e %Zd è un suo
                            divisore\n", n, d);
56
57                      mpz_clears(a, it, d, c, exp, NULL);
58                      gmp_randclear(state);
59
60                      return false;
61                  } 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
62                      mpz_sub_ui(exp, n, 1); // exp = n - 1
63                      exp_mod(c, a, exp, n); // calcoliamo a^(n-1) mod n
64                      if (mpz_cmp_si(c, 1)!=0) {
65                              gmp_printf("%Zd non è un numero primo\n", n);
66
67                              mpz_clears(a, it, d, c, exp, NULL);
68                              gmp_randclear(state);
69
70                              return false;
71                      } else {
72                              RandNumber(a, n, state);
73                              mpz_add_ui(it, it, 1); // it = it + 1
74                      }
75                  }
76          }
77
78          // Pulizia della memoria
79          mpz_clears(a, it, d, c, exp, NULL);
80          gmp_randclear(state);
81
82          return true;
83  }
84
85  void RandNumber(mpz_t a, mpz_t n, gmp_randstate_t state) {
86      mpz_t range;
87      mpz_init(range);
88
89      mpz_sub_ui(range, n, 3);  // range = (n - 2) - 2 + 1 = n - 3
90
91      // Genera un numero casuale in [0, range]
92      mpz_urandomm(a, state, range);
93      // Sposta il numero generato nell'intervallo [2, n-2]
94      mpz_add_ui(a, a, 2);
95
96      mpz_clear(range);
97  }
98
99  // calcola in x la base b elevata alla potenza exp>=0 modulo n (exp binario)
100 void exp_mod (mpz_t x, mpz_t b, mpz_t exp, mpz_t n) {
101         mpz_t squares, e;
102         mpz_init_set(squares, b);
103         mpz_init_set(e, exp);
104         mpz_set_si(x, 1);
105
106         while (mpz_cmp_si(e, 0)>0) { // finchè e > 0
107                 if (mpz_tstbit(e, 0)) { // se bit è 1 moltiplico x per il quadrato
                        (mod n), altrimenti lascio così
108                     mpz_mul(x, x, squares);
109                     mpz_mod(x, x, n);
110                 }
111                 mpz_mul(squares, squares, squares); // aggiorno il quadrato (mod n)
112                 mpz_mod(squares, squares, n); // squares mod n
113                 mpz_fdiv_q_2exp(e, e, 1); // shifto di 1 i bit di e
114         }
115
116         mpz_clears(squares, e, NULL);
117
```

```
118         return;
119 }
```

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

```
1  #include <stdio.h>
2  #include <stdbool.h>
3  #include <string.h>
4  #include <time.h>
5  #include <gmp.h>
6  #include "mcd.h"
7
8  bool eulero (mpz_t, mpz_t);
9  void RandNumber (mpz_t, mpz_t, gmp_randstate_t);
10 void exp_mod (mpz_t, mpz_t, mpz_t, mpz_t);
11
12 int main() {
13         mpz_t n, r, it_max;
14         mpz_inits(n, r, it_max, NULL);
15         printf("n=");
16         gmp_scanf("%Zd", &n);
17
18         // 2 e 3 sono primi
19         if (mpz_cmp_si(n, 2)==0 || mpz_cmp_si(n, 3)==0) {
20                 gmp_printf("%Zd è un numero primo\n", n);
21                 return 0;
22         }
23
24         // i numeri pari diversi da 2 ovviamente non sono primi
25         mpz_fdiv_r_ui(r, n, 2);
26         if (mpz_cmp_si(r, 0)==0) {
27                 gmp_printf("%Zd non è un numero primo e 2 è un suo divisore\n", n);
28                 return 0;
29         }
30
31         // studiamo la primalità dei numeri dispari
32         mpz_set_ui(it_max, mpz_sizeinbase(n, 2));
33         if (eulero(n, it_max)) {
34                 gmp_printf("%Zd è un numero primo\n", n);
35         }
36
37         mpz_clears(n, r, it_max, NULL);
38
39         return 0;
40 }
41
42 bool eulero (mpz_t n, mpz_t it_max) {
43         mpz_t a, it, d, c, exp, n1;
44         mpz_inits(a, it, d, c, exp, n1, NULL);
45         gmp_randstate_t state;
46         gmp_randinit_mt(state);
```

```
47          gmp_randseed_ui(state, time(NULL));
48          mpz_set_si(it, 0);
49          mpz_set_si(a, 2);
50
51          while (mpz_cmp(it, it_max)<0) { // finchè it < it_max
52                  mcd_euclide(d, a, n); // calcolo il massimo comune divisore tra a e
                        n
53                  if (mpz_cmp_si(d, 1)!=0) { // se d!=1, n non è primo e d è un suo
                        divisore
54                          gmp_printf("%Zd non è un numero primo e %Zd è un suo
                                divisore\n", n, d);
55
56                          mpz_clears(a, it, d, c, exp, n1, NULL);
57                          gmp_randclear(state);
58
59                          return false;
60                  } 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
61                          mpz_sub_ui(exp, n, 1);
62                          mpz_fdiv_q_ui(exp, exp, 2); // exp = (n-1)/2
63                          exp_mod(c, a, exp, n); // calcoliamo a^((n-1)/2) mod n
64                          mpz_sub_ui(n1, n, 1); // la classe -1 corrisponde alla
                                classe n-1
65                          if (mpz_cmp_si(c, 1)!=0 && mpz_cmp(c, n1)!=0) {
66                                  gmp_printf("%Zd non è un numero primo\n", n);
67
68                                  mpz_clears(a, it, d, c, exp, n1, NULL);
69                                  gmp_randclear(state);
70
71                                  return false;
72                          } else {
73                                  RandNumber(a, n, state);
74                                  mpz_add_ui(it, it, 1); // it = it + 1
75                          }
76                  }
77          }
78
79          // Pulizia della memoria
80          mpz_clears(a, it, d, c, exp, n1, NULL);
81          gmp_randclear(state);
82
83          return true;
84  }
85
86  void RandNumber(mpz_t a, mpz_t n, gmp_randstate_t state) {
87      mpz_t range;
88      mpz_init(range);
89
90      mpz_sub_ui(range, n, 3);  // range = (n - 2) - 2 + 1 = n - 3
91
92      // Genera un numero casuale in [0, range]
93      mpz_urandomm(a, state, range);
94      // Sposta il numero generato nell'intervallo [2, n-2]
95      mpz_add_ui(a, a, 2);
96
97      mpz_clear(range);
98  }
99
100 // calcola in x la base b elevata alla potenza exp>=0 modulo n (exp binario)
101 void exp_mod (mpz_t x, mpz_t b, mpz_t exp, mpz_t n) {
102         mpz_t squares, e;
103         mpz_init_set(squares, b);
104         mpz_init_set(e, exp);
105         mpz_set_si(x, 1);
106
107         while (mpz_cmp_si(e, 0)>0) { // finchè e > 0
108                 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);
```

```
35
36      for (int i = 0; i < iterations; i++) {
37          mpz_urandomm(a, state, n_minus1);
38          mpz_add_ui(a, a, 2);
39
40          mcd(gcd, a, n);
41          if (mpz_cmp_ui(gcd, 1) != 0) return 0;
42
43          moduloExponentiation(modExp, a, exp, n);
44          int jacobi = jacobiSymbol(a, n);
45          if (jacobi == 0 || mpz_cmp_ui(modExp, (jacobi + mpz_get_ui(n)) % mpz_get_ui
                (n)) != 0) return 0;
46      }
47
48      mpz_clears(a, n_minus1, exp, gcd, modExp, NULL);
49      gmp_randclear(state);
50      return 1;
51  }
52
53  // Main
54  int main() {
55      mpz_t n;
56      int iterations;
57
58      mpz_init(n);
59
60      printf("Inserisci il numero da testare: ");
61      gmp_scanf("%Zd", n);
62
63      printf("Inserisci il numero di iterazioni k: ");
64      scanf("%d", &iterations);
65
66      if (solovayStrassen(n, iterations)) {
67          gmp_printf("%Zd e' molto probabilmente primo.\n", n);
68      } else {
69          gmp_printf("%Zd non e' primo.\n", n);
70      }
71
72      mpz_clear(n);
73      return 0;
74  }
```

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;
```

```
69
70      mpz_t d, a, g, temp;
71      mpz_inits(d, a, g, temp, NULL);
72
73      // Scriviamo n - 1 come 2^h * d
74      mpz_sub_ui(d, n, 1);
75      long h = 0;
76      while (mpz_even_p(d)) {
77          mpz_fdiv_q_2exp(d, d, 1);
78          h++;
79      }
80
81      gmp_randstate_t state;
82      gmp_randinit_mt(state);
83
84      for (int i = 0; i < iterations; i++) {
85          // Genera un numero casuale a tra [2, n-2]
86          mpz_sub_ui(temp, n, 4);
87          mpz_urandomm(a, state, temp);
88          mpz_add_ui(a, a, 2);
89
90          // Controllo MCD
91          mcd(g, a, n);
92          if (mpz_cmp_ui(g, 1) > 0) {
93              mpz_clears(d, a, g, temp, NULL);
94              gmp_randclear(state);
95              return 0;
96          }
97
98          if (isComposite(n, a, d, h)) {
99              mpz_clears(d, a, g, temp, NULL);
100             gmp_randclear(state);
101             return 0;
102         }
103     }
104
105     mpz_clears(d, a, g, temp, NULL);
106     gmp_randclear(state);
107     return 1;
108 }
109
110 int main() {
111     mpz_t n;
112     int iterations;
113
114     mpz_init(n);
115
116     printf("Inserisci un numero da verificare: ");
117     gmp_scanf("%Zd", n);
118
119     printf("Inserisci il numero di iterazioni del test: ");
120     scanf("%d", &iterations);
121
122     if (millerRabin(n, iterations)) {
123         gmp_printf("%Zd è molto probabilmente primo.\n", n);
124     } else {
125         gmp_printf("%Zd è composto.\n", n);
126     }
127
128     mpz_clear(n);
129     return 0;
130 }
```

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)
```

```
47                  funz_f(f, x, n);
48                  funz_g(g, y, n);
49
50                  // calcoliamo f(x)-g(x) mod n
51                  mpz_sub(x_new, g, f); // x_new = g - f
52                  mpz_mod(x_new, x_new, n); // x_new = x_new mod n
53                  mcd_euclide(d, x_new, n); // d = MCD(x_new, n)
54
55                  if (mpz_cmp_si(d, 1)!=0 && mpz_cmp(d, n)!=0) { // se d è diverso da
                        1 ho trovato un divisore di n
56                      return true;
57                  }
58
59                  mpz_set(x, f); // la nuova x è f(x) calcolato prima
60                  mpz_set(y, g); // la nuova y è g(y) calcolato prima
61                  mpz_add_ui(it, it, 1); // it = it + 1
62          }
63
64          // Pulizia della memoria
65          mpz_clears(x, y, f, g, x_new, it, NULL);
66
67          return false;
68  }
69
70  void funz_f (mpz_t result, mpz_t x, mpz_t n) {
71          mpz_t temp;
72          mpz_init(temp);
73
74          mpz_mul(temp, x, x);  // temp = x * x
75          mpz_add_ui(result, temp, 1);  // result = temp + 1
76
77          mpz_mod (result, result, n); // calcolo result mod n
78
79          // Pulizia memoria
80          mpz_clear(temp);
81  }
82
83  void funz_g (mpz_t result, mpz_t x, mpz_t n) {
84          mpz_t temp, temp2, temp4;
85          mpz_inits(temp, temp2, temp4, NULL);
86
87          mpz_mul(temp2, x, x);  // temp2 = x * x (calcolo x^2)
88          mpz_mul(temp4, temp2, temp2);  // temp4 = temp2 * temp2 (calcolo x^4)
89          mpz_mul_ui(temp2, temp2, 2);  // temp2 = temp2 * 2 (calcolo 2*x^2)
90          mpz_add(temp, temp4, temp2);  // temp = temp4 + temp2 (calcolo x^4+2*x^2)
91          mpz_add_ui(result, temp, 2);  // result = temp + 1 (calcolo x^4+2*x^2+1)
92
93          mpz_mod (result, result, n); // calcolo result mod n
94
95          // Pulizia memoria
96          mpz_clears(temp, temp2, temp4, NULL);
97  }
```

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)
```

```
52      unsigned long int base_length = mpz_get_ui(bound);
53      base_length=(base_length-1)/2 +2;
54      mpz_t *base = (mpz_t*)malloc(base_length*sizeof(mpz_t));
55      mpz_init_set_si(base[0],-1);
56      mpz_init_set_si(base[1],2);
57      for (i=2; i<base_length; i++) {
58          mpz_init_set_si(base[i],2*i-1);
59      }
60
61      // 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]
62      int** M=(int**)malloc((base_length+1)*sizeof(int*));
63      for (i=0; i<base_length+1; i++) M[i]=(int*)malloc(base_length*sizeof(int));
64      for (i=0; i<base_length+1; i++) for (j=0; j<base_length; j++) M[i][j]=0; //
            inizializzo a 0
65
66      // dati iniziali delle relazioni ricorsive per lo sviluppo in frazione continua
             di radice di n
67      mpz_t a,beta,gamma,bmeno1,bmeno2;
68      mpz_inits(a,beta,gamma,bmeno1,bmeno2,NULL);
69      mpz_sqrt(a,n);
70      mpz_neg(beta,a);
71      mpz_set_ui(gamma,1);
72      mpz_set_ui(bmeno1,1);
73      mpz_set_ui(bmeno2,0);
74
75      unsigned long int it=0;
76      while (it<iter) {
77          // costruisco i b_i con quadrato piccolo mod n: ne voglio base_length+1
                cosi' da assicurarmi almeno una combinazione lineare di righe
78          // nulla nella matrice che costruiro' dopo
79          mpz_t* b=cont_frac_sqrt_mod(n,bmeno1,bmeno2,a,beta,gamma,base_length+1);
80          mpz_t* squares=squares_mod(b,base_length+1,n);
81
82          // calcolo la matrice degli esponenti massimi dei primi che dividono
                ciascun quadrato
83          for (i=0; i<base_length+1; i++) {
84              // primo termine (corrisponde a -1 nella base)
85              if (mpz_cmp_si(squares[i],0)<0) {
86                  M[i][0]=1;
87                  mpz_neg(squares[i],squares[i]);
88              }
89
90              // secondo termine (corrisponde a 2 nella base)
91              while (mpz_divisible_ui_p(squares[i],2)) {
92                  mpz_divexact_ui(squares[i],squares[i],2);
93                  M[i][1]++;
94              }
95
96              // termini rimanenti (corrispondono ai dispari (2*j -1) nella base)
97              for (j=2; j<base_length; j++) {
98                  while (mpz_divisible_ui_p(squares[i],2*j-1)) {
99                      mpz_divexact_ui(squares[i],squares[i],2*j-1);
100                     M[i][j]++;
101                 }
102             }
103
104             // controllo se il quadrato e' stato completamente scomposto nella base
                    ; in caso contrario lo sostituisco con un nuovo b
105             // (e il suo quadrato) e ritento la scomposizione
106             if (mpz_cmp_si(squares[i],1)!=0) {
107                 cont_frac_next_term(b[i],n,bmeno1,bmeno2,a,beta,gamma);
108                 mpz_mul(squares[i],b[i],b[i]);
109                 riduci_mod_min(squares[i],n);
110                 i--; // attenzione: in questo modo il ciclo for su i potrebbe non
                        terminare: aggiungiamo una qualche condizione?
111             }
112         }
113
114         // conto le colonne nulle poiche' possono essere ignorate nell'eliminazione
```

```
            di Gauss, facendo risparmiare spazio e tempo
115         unsigned long int num_null_cols;
116         bool* is_col_zeros=(bool*)malloc(base_length*sizeof(bool));
117         num_null_cols = check_null_column(is_col_zeros,M,base_length+1,base_length)
                ;
118         unsigned long int num_cols_M_mod2 = base_length-num_null_cols;
119         // creo un vettore di "differenze cumulative", lungo come il numero di
                colonne di M_mod2, che tiene conto di quante colonne sono state
120         // 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,
121         // e cioe' all'elemento della base di indice (j+cumul_diff[j]).
122         int* cumul_diff=(int*)malloc(num_cols_M_mod2*sizeof(int));
123         int contatore=0;
124         i=0; j=0;
125         while (j<num_cols_M_mod2) {
126             while (is_col_zeros[i]) {
127                 contatore++;
128                 i++;
129                 if (i>base_length) break;
130             }
131             cumul_diff[j]=contatore;
132             i++; j++;
133         }
134         // creo la matrice booleana M_mod2 che contiene i coefficienti di M ridotti
                modulo 2, escluse le colonne nulle (true=1, false=0).
135         // sara' una matrice di dimensioni (base_length+1)*(base_length-
                num_null_cols).
136         bool** M_mod2=(bool**)malloc((base_length+1)*sizeof(bool*));
137         for (i=0; i<base_length+1; i++) M_mod2[i]=(bool*)malloc(num_cols_M_mod2*
                sizeof(bool));
138         for (i=0; i<base_length+1; i++) {
139             for (j=0; j<num_cols_M_mod2; j++) {
140                 M_mod2[i][j]=M[i][j+cumul_diff[j]] % 2;
141             }
142         }
143
144         // costruisco la matrice identità da affiancare a M_mod2 prima di applicare
                Gauss
145         bool** Id=(bool**)malloc((base_length+1)*sizeof(bool*));
146         for (i=0; i<base_length+1; i++) Id[i]=(bool*)malloc((base_length+1)*sizeof(
                bool));
147         for (i=0; i<base_length+1; i++) for (j=0; j<base_length+1; j++) Id[i][j]=
                false; // inizializzo a 0
148         for (i=0; i<base_length+1; i++) Id[i][i]=true; // pongo la diagonale uguale
                ad 1
149
150         // applico Gauss e calcolo A e B
151         Gauss(M_mod2, Id, base_length+1, num_cols_M_mod2);
152         unsigned long int index=0;
153         unsigned long int null_row=0;
154         mpz_t A, B, menoB, ApiuB;
155             mpz_inits(A, B, menoB, ApiuB, NULL);
156         while (index<(base_length+1)) {
157                     null_row=find_null_rows(M_mod2, base_length+1, num_cols_M_mod2,
                            index);
158                     get_B(B, n, Id, b, cumul_diff, null_row, (base_length+1));
159                         get_A(A, n, base, M, base_length, Id, cumul_diff, null_row)
                                ;
160             mpz_sub(menoB,n,B);
161             mpz_fdiv_r(menoB,menoB,n);
162
163                     if (mpz_cmp(A, B)==0 || mpz_cmp(A, menoB)==0) { // se A=B
                            oppure A=-B mod n, l'algoritmo è fallito
164                         if (index!=base_length) { // se non abbiamo esaurito le
                                righe riproviamo con un'altra combinazione
165                             index++;
166                         } else { // altrimenti ricominciamo da capo
167                                 break; // in questo modo rientriamo nel ciclo while
                                    (it<iter)
168                         }
```

```
169                     } else { // se invece se A!=B oppure A!=-B mod n concludo l'
                            algoritmo calcolando d=MCD(A+B, n)
170                             mpz_add(ApiuB, A, B);
171                             mcd_euclide(d, ApiuB, n);
172
173 /*                if (mpz_cmp_si(d,1)==0) {//// sarebbe da rimuovere; e' solo per
        evitare che l'algoritmo (sbagliato) restituisca 1
174                     index++;
175                     continue;
176                 }
177 */
178
179                 // pulizia memoria
180                             mpz_clears(A, B, menoB, ApiuB, NULL);
181                 free(is_col_zeros); free(cumul_diff);
182                 for (i=0; i<base_length+1; i++) free(M_mod2[i]); free(M_mod2);
183                 for (i=0; i<base_length+1; i++) free(Id[i]); free(Id);
184                 for (i=0; i<base_length+1; i++) mpz_clear(b[i]); free(b);
185                 for (i=0; i<base_length+1; i++) mpz_clear(squares[i]); free(squares
                        );
186                 for (i=0; i<base_length+1; i++) free(M[i]); free(M);
187                 for (i=0; i<base_length; i++) mpz_clear(base[i]); free(base);
188                 mpz_clears(a,beta,gamma,bmeno1,bmeno2,NULL);
189                             return 1;
190                 }
191                 }
192         // pulizia memoria
193                 mpz_clears(A, B, menoB, ApiuB, NULL);
194         free(is_col_zeros); free(cumul_diff);
195         for (i=0; i<base_length+1; i++) free(M_mod2[i]); free(M_mod2);
196         for (i=0; i<base_length+1; i++) free(Id[i]); free(Id);
197         for (i=0; i<base_length+1; i++) mpz_clear(b[i]); free(b);
198         for (i=0; i<base_length+1; i++) mpz_clear(squares[i]); free(squares);
199         it++;
200     }
201
202     // se sono arrivato qui ho completato tutte le iterazioni senza trovare un
            divisore: fallimento.
203
204     for (i=0; i<base_length+1; i++) free(M[i]); free(M);
205     for (i=0; i<base_length; i++) mpz_clear(base[i]); free(base);
206     mpz_clears(a,beta,gamma,bmeno1,bmeno2,NULL);
207     return 0;
208 }
209
210 // Restituisce k>0 numeratori dell'approssimazione in frazione continua di sqrt(n)
        modulo n, espressi con valore assoluto minimo, a partire
211 // dai due numeratori precedenti e dai corrispondenti valori di a, beta, gamma (che
         vengono modificati durante l'esecuzione).
212 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) {
213     int i;
214     mpz_t sqrt_n,temp;
215     mpz_inits(sqrt_n,temp,NULL);
216     mpz_sqrt(sqrt_n,n);
217     mpz_t* b=(mpz_t*)malloc(k*sizeof(mpz_t));
218     for (i=0;i<k;i++) mpz_init(b[i]);
219
220     // esplicito i primi due termini (poiché utilizzano dati iniziali specifici su
            b_-1 e b_-2)
221
222     mpz_mul(temp,a,bmeno1);
223     mpz_add(b[0],temp,bmeno2); // aggiornato b[0]
224     mpz_mul(temp,beta,beta);
225     mpz_sub(temp,n,temp);
226     mpz_fdiv_q(gamma,temp,gamma); // aggiornato gamma
227     mpz_sub(temp,sqrt_n,beta);
228     mpz_fdiv_q(a,temp,gamma); // aggiornato a
229     mpz_mul(temp,a,gamma);
230     mpz_add(temp,temp,beta);
```

```
231     mpz_neg(beta,temp); // aggiornato beta
232     riduci_mod_min(b[0],n);
233 
234     if (k==1) {
235         mpz_set(bmeno2,bmeno1);
236         mpz_set(bmeno1,b[0]);
237         return b;
238     }
239 
240     mpz_mul(temp,a,b[0]);
241     mpz_add(b[1],temp,bmeno1); // aggiornato b[1]
242     mpz_mul(temp,beta,beta);
243     mpz_sub(temp,n,temp);
244     mpz_fdiv_q(gamma,temp,gamma); // aggiornato gamma
245     mpz_sub(temp,sqrt_n,beta);
246     mpz_fdiv_q(a,temp,gamma); // aggiornato a
247     mpz_mul(temp,a,gamma);
248     mpz_add(temp,temp,beta);
249     mpz_neg(beta,temp); // aggiornato beta
250     riduci_mod_min(b[1],n);
251 
252     if (k==2) {
253         mpz_set(bmeno2,b[0]);
254         mpz_set(bmeno1,b[1]);
255         return b;
256     }
257 
258     for (i=2; i<k; i++) {
259         mpz_mul(temp,a,b[i-1]);
260         mpz_add(b[i],temp,b[i-2]); // aggiornato b[i]
261         mpz_mul(temp,beta,beta);
262         mpz_sub(temp,n,temp);
263         mpz_fdiv_q(gamma,temp,gamma); // aggiornato gamma
264         mpz_sub(temp,sqrt_n,beta);
265         mpz_fdiv_q(a,temp,gamma); // aggiornato a
266         mpz_mul(temp,a,gamma);
267         mpz_add(temp,temp,beta);
268         mpz_neg(beta,temp); // aggiornato beta
269         riduci_mod_min(b[i],n);
270     }
271 
272     mpz_set(bmeno2,b[k-2]);
273     mpz_set(bmeno1,b[k-1]);
274 
275     mpz_clears(sqrt_n,temp,NULL);
276     return b;
277 }
278 
279 // salva in b il termine successivo dello sviluppo in frazione continua di sqrt(n)
        modulo n, espresso con valore assoluto minimo, a partire
280 // dai due numeratori precedenti e dai corrispondenti valori di a, beta, gamma (che
         vengono modificati durante l'esecuzione).
281 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) {
282     mpz_t sqrt_n,temp;
283     mpz_inits(sqrt_n,temp,NULL);
284     mpz_sqrt(sqrt_n,n);
285 
286     mpz_mul(temp,a,bmeno1);
287     mpz_add(b,temp,bmeno2); // aggiornato b
288     mpz_mul(temp,beta,beta);
289     mpz_sub(temp,n,temp);
290     mpz_fdiv_q(gamma,temp,gamma); // aggiornato gamma
291     mpz_sub(temp,sqrt_n,beta);
292     mpz_fdiv_q(a,temp,gamma); // aggiornato a
293     mpz_mul(temp,a,gamma);
294     mpz_add(temp,temp,beta);
295     mpz_neg(beta,temp); // aggiornato beta
296     riduci_mod_min(b,n);
297 
```

```
298     mpz_set(bmeno2,bmeno1);
299     mpz_set(bmeno1,b); // aggiornati i termini precedenti
300 
301     mpz_clears(sqrt_n,temp,NULL);
302     return;
303 }
304 
305 // restituisce un array con i quadrati degli elementi nell'array in input (lungo k)
     modulo n, espressi con valore assoluto minimo.
306 mpz_t* squares_mod(mpz_t* array, unsigned long int k, mpz_t n) {
307     int i;
308     mpz_t* squares=(mpz_t*)malloc(k*sizeof(mpz_t));
309     for (i=0;i<k;i++){
310         mpz_init(squares[i]);
311         mpz_mul(squares[i],array[i],array[i]);
312         riduci_mod_min(squares[i],n);
313     }
314 
315     return squares;
316 }
317 
318 // riduce a modulo n esprimendolo con valore assoluto minimo (potra' essere
    positivo o negativo, ma avra' abs(a)<= n/2).
319 void riduci_mod_min(mpz_t a, mpz_t n){
320     mpz_t n_mezzi;
321     mpz_init(n_mezzi);
322     mpz_fdiv_q_2exp(n_mezzi,n,1);
323 
324     mpz_fdiv_r(a,a,n);
325     if (mpz_cmp(a,n_mezzi)>0) mpz_sub(a,a,n); // se a>n/2, allora a-->a-n
326 
327     mpz_clear(n_mezzi);
328     return;
329 }
330 
331 // 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.
332 unsigned long int check_null_column(bool* is_zero, int** M, unsigned long int m,
    unsigned long int n) {
333     int i,j;
334     unsigned long int num_null_columns=0;
335     for (j=0; j<n; j++) {
336         is_zero[j]=true;
337         num_null_columns++;
338         for (i=0; i<m; i++) {
339             if (M[i][j]!=0) {
340                 is_zero[j]=false;
341                 num_null_columns--;
342                 break;
343             }
344         }
345     }
346     return num_null_columns;
347 }
348 
349 //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
350 void Gauss(bool **M, bool **Id, unsigned long int rows, unsigned long int cols) {
351     int row=0;
352     for (int col=0; col<cols && row<rows; col++) {
353         // Trova il pivot nella colonna corrente
354         int pivot=-1;
355         for (int i=row; i<rows; i++) {
356             if (M[i][col]) {
357                 pivot=i;
358                 break;
359             }
360         }
361 
```

```
362         // Se non troviamo un pivot, passiamo alla colonna successiva
363         if (pivot==-1) {
364                 continue;
365         }
366
367         // Scambia la riga pivot con la riga attuale
368         if (pivot!=row) {
369             bool *temp=M[pivot];
370             M[pivot]=M[row];
371             M[row]=temp;
372
373             temp=Id[pivot];
374             Id[pivot]=Id[row];
375             Id[row]=temp;
376         }
377
378         // Elimina i valori nelle altre righe (comando XOR)
379         for (int i=0; i<rows; i++) {
380             if (i!=row && M[i][col]) {
381                 for (int j=0; j<cols; j++) {
382                     M[i][j] ^= M[row][j];
383 //                    Id[i][j] ^= Id[row][j];
384                 }
385                 for (int j=0; j<rows; j++) { // forse devo fare cosi'? Il fatto e'
                        che Id ha piu' colonne di M
386                     Id[i][j] ^= Id[row][j];
387                 }
388             }
389         }
390
391         // Passa alla prossima riga
392         row++;
393     }
394     return;
395 }
396
397 // troviamo la prima riga nulla di M_mod2 dopo aver applicato Gauss, l'indice
        indica da quela riga cominciare a fare la ricerca
398 unsigned long int find_null_rows (bool** mat, unsigned long int rows, unsigned long
        int cols, unsigned long int index) {
399         unsigned long int null_row=rows; // inizializzo ad un intero che non può
            assumere
400         for (int i=index; i<rows; i++) {
401                 for (int j=0; j<cols; j++) {
402                         if (mat[i][j]==true) { // appena incontro un 1 interrompo
                            il ciclo sulle colonne
403                             break;
404                         } else if (j==cols-1 && mat[i][j]==false) { // se ho
                            controllato tutte le colonne modifico null_row
405                             null_row=i;
406                         }
407                 }
408                 if (null_row!=rows) { // se ho modificato null_row ho trovato una
                        riga nulla, posso interrompere la ricerca
409                     break;
410                 }
411         }
412         return null_row;
413 }
414
415 // 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
416 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) {
417         mpz_set_si(B, 1);
418         for (int j=0; j<cols; j++){
419                 if (Id[null_row][j]==true) {
420 //                    mpz_mul(B, B, b[j+cumul_diff[j]]);
421             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,
```

```
422                                 // ossia il j-esimo elemento dei b; giusto? Anche
                                        perche' get_B viene chiamata con cols=
                                        base_length+1, e b e'
423                                 // lungo base_length+1, cosi' come la dimensione
                                        di Id
424
425                         mpz_fdiv_r(B,B,n);
426                 }
427         }
428         return;
429 }
430
431 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) {
432         mpz_set_si(A, 1);
433         // calcolo exp_tot, ossia sommo le righe i di M (i sono quelli ricavati da
                gauss)
434         int* exp_tot=(int*)malloc(base_length*sizeof(int));
435         for (int i=0; i<base_length; i++) exp_tot[i]=0; // inizializzo a zero
436         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)
437                 if (Id[null_row][i]==true) { // allora consideriamo la riga i+
                        cumul_diff[i] di M
438                         for (int j=0; j<base_length; j++) {
439 //                              exp_tot[j]=exp_tot[j]+M[i+cumul_diff[i]][j];
440                 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
441                         }
442                 }
443         }
444
445         // ora calcoliamo A=base[i]^(exp_tot[i]/2)
446         mpz_t temp;
447         mpz_inits(temp, NULL);
448         for (int i=0; i<base_length; i++) {
449                 mpz_powm_ui(temp, base[i], exp_tot[i]/2, n);
450                 mpz_mul(A, A, temp);
451                 mpz_fdiv_r(A,A,n);
452         }
453
454         mpz_clears(temp, NULL);
455         free(exp_tot);
456
457         return;
458 }
```

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

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

```
3  #include <string.h>
4  #include <math.h>
5  #include <gmp.h>
6  #include "..\Include\mcd.h"
7
8  bool p1_pollard (mpz_t, mpz_t);
9  void exp_mod (mpz_t, mpz_t, mpz_t, mpz_t);
10
11 int main () {
12         mpz_t n, d, r;
13         mpz_inits(n, d, r, NULL);
14         printf("Inserire il numero non primo da fattorizzare: n = ");
15         gmp_scanf("%Zd", n);
16
17         mpz_fdiv_r_ui (r, n, 2);
18         if (mpz_cmp_si(r,0)==0) { // se il numero è pari allora 2 è un divisore
               proprio
19                 gmp_printf("Un divisore di %Zd è d = 2\n", n);
20         } else {
21                 if (p1_pollard(d, n)) {
22                         gmp_printf("Un divisore di %Zd è d = %Zd\n", n, d);
23                 } else {
24                         gmp_printf("Non è stato trovato un divisore di %Zd\n", n);
25                 }
26         }
27
28         // Pulizia della memoria
29         mpz_clears(n, d, r, NULL);
30
31         return 0;
32 }
33
34 bool p1_pollard (mpz_t d, mpz_t n) {
35         mpz_t b, b0, b1, a, m, y, it;
36         mpz_inits(b, b0, b1, a, m, y, it, NULL);
37         mpz_set_si(a, 2);
38         mpz_set_si(m, 1);
39         mpz_set_si(b, 2);
40         int primes[18] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47,
               53, 59, 61};
41
42         while (mpz_cmp(b, n)<0) {
43                 // cerco m tc m sia la produttoria su i di p_i^e_i con p_i primi e
                       p_i^e_i < b
44                 // tuttavia essendo la base di primi limitata, da 61 in poi tutti
                       per tutti i primi è come se prendessimo e_i=0
45                 for (int i=0; i<18; i++) {
46                         if (mpz_cmp_si(b, primes[i])<0) { // appena b > primes[i]
                               interrompo il for
47                                 break;
48                         } else { // finchè b < primes(i)
49                                 int e;
50                                 e=pow(primes[i], strlen(mpz_get_str(NULL,primes[i],
                                       b))-1);
51                                 mpz_mul_ui(m, m, e);  // m = m * e
52                         }
53                 }
54
55                 exp_mod(y, a, m, n);
56
57                 mpz_sub_ui(y, y, 1);
58
59                 mcd_euclide(d, y, n); // d = MCD(a^m-1, n)
60                 if (mpz_cmp_si(d, 1)!=0 && (mpz_cmp(d, n)!=0)) {
61                         return true;
62                 }
63
64                 mpz_add_ui(b, b, 1); // it = it + 1
65         }
66
```

```
67          mpz_clears(b, b0, b1, a, m, y, it, NULL);
68
69          return false;
70  }
71
72  // calcola in x la base b elevata alla potenza exp>=0 modulo n (exp binario)
73  void exp_mod (mpz_t x, mpz_t b, mpz_t exp, mpz_t n) {
74          mpz_t squares, e;
75          mpz_init_set(squares, b);
76          mpz_init_set(e, exp);
77          mpz_set_si(x, 1);
78
79          while (mpz_cmp_si(e, 0)>0) { // finchè e > 0
80                  if (mpz_tstbit(e, 0)) { // se bit è 1 moltiplico x per il quadrato
                        (mod n), altrimenti lascio così
81                          mpz_mul(x, x, squares);
82                          mpz_mod(x, x, n);
83                  }
84                  mpz_mul(squares, squares, squares); // aggiorno il quadrato (mod n)
85                  mpz_mod(squares, squares, n); // squares mod n
86                  mpz_fdiv_q_2exp(e, e, 1); // shifto di 1 i bit di e
87          }
88
89          mpz_clears(squares, e, NULL);
90
91          return;
92  }
```

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

```
1   #include <string.h>
2   #include <time.h>
3   #include "mcd.h"
4   #include "Bezout.h"
5
6   // 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)
7   #define NUM_PRIMES 4
8
9   int lenstra (mpz_t, mpz_t);
10  int lenstra_iter (mpz_t, mpz_t);
11
12  // funzioni ausiliarie per algoritmo di Lenstra
13  int double_elliptic (mpz_t, mpz_t, mpz_t, mpz_t, mpz_t, mpz_t, mpz_t);
14  int sum_elliptic (mpz_t, mpz_t, mpz_t, mpz_t, mpz_t, mpz_t, mpz_t, mpz_t);
15  int multiply_elliptic (mpz_t, mpz_t, mpz_t, mpz_t, mpz_t, int, mpz_t, mpz_t);
16  void hasse (mpz_t, mpz_t);
17  void create_exp_primes (unsigned int*, int*, mpz_t);
18  void calcola_b (mpz_t, mpz_t, mpz_t, mpz_t, mpz_t);
19  void calcola_delta (mpz_t, mpz_t, mpz_t, mpz_t);
20
21  // Fattorizza l'intero positivo n utilizzando l'algoritmo di Lenstra basato su
        curve ellittiche: se riesce restituisce 1 e salva in
```

```
22 // d un divisore proprio di n, altrimenti restituisce 0. Utilizza coefficienti e punti sulle curve ellittiche generati casualmente, ed esegue
23 // un numero di iterazioni fissato all'interno della funzione (attualmente 2^30).
24 int lenstra (mpz_t d, mpz_t n) {
25
26     // salvo una lista di primi da utilizzare nel tentativo di fattorizzare n: scelgo i primi 18 poiché la funzione mpz_get_str, che uso
27     // 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
28     // può aumentare, ma bisogna trovare un modo alternativo per calcolare la stima, e in ogni caso solitamente bastano pochi primi.
29     int primes[18] = {2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61};
30
31     mpz_t r;
32     mpz_init(r);
33     // faccio due prime divisioni test: nell'algoritmo serve n dispari, e sul libro di Koblitz chiede che i primi che dividano n siano >3.
34     mpz_fdiv_r_ui(r,n,2);
35     if (mpz_cmp_si(r,0)==0) {
36         mpz_set_ui(d,2);
37         mpz_clear(r);
38         return 1;
39     }
40     mpz_fdiv_r_ui(r,n,3);
41     if (mpz_cmp_si(r,0)==0) {
42         mpz_set_ui(d,3);
43         mpz_clear(r);
44         return 1;
45     }
46     mpz_clear(r);
47
48     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
49     mpz_inits(hasse_n,x_P,y_P,a,b,delta,iter,maxiter,NULL);
50     unsigned int exp_primes[NUM_PRIMES];
51     hasse(hasse_n,n);
52     create_exp_primes(exp_primes,primes,hasse_n); // inizializzo gli esponenti a cui elevare ciascun primo
53     mpz_clear(hasse_n);
54
55     gmp_randstate_t randstate;
56     gmp_randinit_default(randstate);
57     unsigned long int t = time(NULL);
58     gmp_randseed_ui(randstate, t);
59
60     // inizio l'algoritmo di Lenstra
61
62     mpz_set_ui(maxiter,1);
63     mpz_mul_2exp(maxiter,maxiter,30); // testo 2^30 iterazioni
64
65     while (mpz_cmp(iter,maxiter)<0) {
66
67         mpz_urandomm(x_P,randstate,n);
68         mpz_urandomm(y_P,randstate,n);
69         mpz_urandomm(a,randstate,n);
70
71         calcola_b(b,x_P,y_P,a,n); // calcolo b affinché P sia sulla curva ellittica
72         calcola_delta(delta,a,b,n); // calcolo discriminante della curva ellittica (mod n)
73
74         if (mpz_cmp_si(delta,0)==0) goto retry; // se delta=0 cambio a e riprovo
75
76         mcd_binario(d,delta,n);
77
78         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)
79             gmp_randclear(randstate);
80             mpz_clears(x_P,y_P,a,b,delta,iter,maxiter,NULL);
81             return 1;
82         }
```

```
83
84          // inizio i calcoli P --> m*P
85
86          int i,j,outcome;
87          for (i=0; i<exp_primes[0]; i++) { // calcolo P--> 2^(e_2)*P
88
89              outcome=double_elliptic(d,x_P,y_P,x_P,y_P,a,n);
90              if (outcome==-1) goto retry; // se non posso raddoppiare cambio
                    parametri e riprovo
91              if (outcome==0) { // successo: ho trovato in d un divisore proprio di n
92                  gmp_randclear(randstate);
93                  mpz_clears(x_P,y_P,a,b,delta,iter,maxiter,NULL);
94                  return 1;
95              }
96          }
97
98          // calcolo le potenze con i primi della lista maggiori di 2
99          for (j=1; j<NUM_PRIMES; j++) {
100             for (i=0; i<exp_primes[j]; i++) { // calcolo P-->k^(e_k)*P, con k primo
101
102                 outcome=multiply_elliptic(d,x_P,y_P,x_P,y_P,primes[j],a,n);
103                 if (outcome==-1) goto retry; // se non posso moltiplicare cambio
                        parametri e riprovo
104                 if (outcome==0) { // successo: ho trovato in d un divisore proprio
                        di n
105                     gmp_randclear(randstate);
106                     mpz_clears(x_P,y_P,a,b,delta,iter,maxiter,NULL);
107                     return 1;
108                 }
109             }
110         }
111
112         // se arrivo qui ho calcolato mP senza trovare divisori: cambio parametri e
                riprovo
113
114         retry: mpz_add_ui(iter,iter,1); // aggiorno l'indice di iterazione e cambio
                parametri
115     }
116
117     // se arrivo qui non ho trovato divisori di n nel numero di iterazioni fissate:
            fallimento
118
119     gmp_randclear(randstate);
120     mpz_clears(x_P,y_P,a,b,delta,iter,maxiter,NULL);
121     return 0;
122 }
123
124 // Fattorizza l'intero positivo n utilizzando l'algoritmo di Lenstra basato su
        curve ellittiche: se riesce restituisce 1 e salva in
125 // d un divisore proprio di n, altrimenti restituisce 0. Versione iterativa:
        considera in ordine tutti i possibili punti e coefficienti
126 // per le curve ellittiche.
127 int lenstra_iter (mpz_t d, mpz_t n) {
128
129     // salvo una lista di primi da utilizzare nel tentativo di fattorizzare n:
            scelgo i primi 18 poiché la funzione mpz_get_str, che uso
130     // 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
131     // può aumentare, ma bisogna trovare un modo alternativo per calcolare la stima
            , e in ogni caso solitamente bastano pochi primi.
132     int primes[18] = {2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61};
133
134     mpz_t r;
135     mpz_init(r);
136     // faccio due prime divisioni test: nell'algoritmo serve n dispari, e sul libro
            di Koblitz chiede che i primi che dividano n siano >3.
137     mpz_fdiv_r_ui(r,n,2);
138     if (mpz_cmp_si(r,0)==0) {
139         mpz_set_ui(d,2);
140         mpz_clear(r);
```

```
141         return 1;
142     }
143     mpz_fdiv_r_ui(r,n,3);
144     if (mpz_cmp_si(r,0)==0) {
145         mpz_set_ui(d,3);
146         mpz_clear(r);
147         return 1;
148     }
149     mpz_clear(r);
150 
151     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
152     mpz_inits(hasse_n,x_P,y_P,x_iter,y_iter,a,b,delta,NULL);
153     unsigned int exp_primes[NUM_PRIMES];
154     hasse(hasse_n,n);
155     create_exp_primes(exp_primes,primes,hasse_n); // inizializzo gli esponenti a
        cui elevare ciascun primo
156     mpz_clear(hasse_n);
157 
158     // inizio l'algoritmo di Lenstra
159 
160     // (ciclo su x_iter e y_iter poiché nel corso dell'algoritmo x_P e y_P vengono
        modificati)
161     // ciclo sulla componente x del punto P da 0 a n-1
162     while (mpz_cmp(x_iter,n)<0) {
163         mpz_set_ui(y_iter,1);
164 
165         // ciclo sulla componente y del punto P da 1 a n-1 (parte da 1 altrimenti
            il primo raddoppio non è definito)
166         while (mpz_cmp(y_iter,n)<0) {
167             mpz_set_ui(a,0);
168 
169             // ciclo sulla scelta di a tra 0 e n-1
170             while (mpz_cmp(a,n)<0) {
171 
172                 mpz_set(x_P,x_iter);
173                 mpz_set(y_P,y_iter);
174 
175                 calcola_b(b,x_P,y_P,a,n); // calcolo b affinché P sia sulla curva
                    ellittica
176                 calcola_delta(delta,a,b,n); // calcolo discriminante della curva
                    ellittica (mod n)
177 
178                 if (mpz_cmp_si(delta,0)==0) goto retry; // se delta=0 cambio a e
                    riprovo
179 
180                 mcd_binario(d,delta,n);
181 
182                 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)
183                     mpz_clears(x_P,y_P,x_iter,y_iter,a,b,delta,NULL);
184                     return 1;
185                 }
186 
187                 // inizio i calcoli P --> m*P
188 
189                 int i,j,outcome;
190                 for (i=0; i<exp_primes[0]; i++) { // calcolo P--> 2^(e_2)*P
191 
192                     outcome=double_elliptic(d,x_P,y_P,x_P,y_P,a,n);
193                     if (outcome==-1) goto retry; // se non posso raddoppiare cambio
                         a e riprovo
194                     if (outcome==0) {
195                         mpz_clears(x_P,y_P,x_iter,y_iter,a,b,delta,NULL);
196                         return 1; // successo: ho trovato in d un divisore proprio
                            di n
197                     }
198                 }
199 
```

```
200                 // calcolo le potenze con i primi della lista maggiori di 2
201                 for (j=1; j<NUM_PRIMES; j++) {
202                     for (i=0; i<exp_primes[j]; i++) { // calcolo P-->k^(e_k)*P, con
                            k primo
203
204                         outcome=multiply_elliptic(d,x_P,y_P,x_P,y_P,primes[j],a,n);
205                         if (outcome==-1) goto retry; // se non posso moltiplicare
                                cambio a e riprovo
206                         if (outcome==0) {
207                             mpz_clears(x_P,y_P,x_iter,y_iter,a,b,delta,NULL);
208                             return 1; // successo: ho trovato in d un divisore
                                    proprio di n
209                         }
210                     }
211                 }
212
213                 // se arrivo qui ho calcolato mP senza trovare divisori: cambio a e
                        riprovo
214
215                 retry: mpz_add_ui(a,a,1); // sommo 1 ad a e riprovo
216             }
217
218             mpz_add_ui(y_iter,y_iter,1); // sommo 1 a y e riprovo
219         }
220
221         mpz_add_ui(x_iter,x_iter,1); // sommo 1 a x e riprovo
222     }
223
224     // se arrivo qui non ho trovato divisori di n per alcun P e alcun a:
            restituisco 0
225
226     mpz_clears(x_P,y_P,x_iter,y_iter,a,b,delta,NULL);
227     return 0;
228 }
229
230 // 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
231 // 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
232 // proprio d di n, e restituisco 0.
233 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) {
234     mpz_t y_P2_inv,temp1;
235     mpz_inits(y_P2_inv,temp1,NULL);
236     mpz_mul_si(temp1,y_P,2);
237     inv_mod_mcd(d,y_P2_inv,temp1,n);
238     if (mpz_cmp(d,n)==0) { // fallimento: restituisco -1
239         mpz_clears(y_P2_inv,temp1,NULL);
240         return -1;
241     }
242     if (mpz_cmp_si(d,1)!=0) { // successo: restituisco 0
243         mpz_clears(y_P2_inv,temp1,NULL);
244         return 0;
245     }
246     // calcolo 2P: vedi Koblitz per le formule
247     mpz_t temp2,x_P_copy,y_P_copy; // creo copie per non modificare x_P e y_P nei
            calcoli
248     mpz_inits(temp2,x_P_copy,y_P_copy,NULL);
249     mpz_set(x_P_copy,x_P); mpz_set(y_P_copy,y_P);
250     mpz_mul(temp1,x_P_copy,x_P_copy);
251     mpz_mul_si(temp1,temp1,3);
252     mpz_add(temp1,temp1,a);
253     mpz_fdiv_r(temp1,temp1,n);
254     mpz_mul(temp1,temp1,y_P2_inv);
255     mpz_fdiv_r(temp1,temp1,n);
256     mpz_mul(x_2P,temp1,temp1);
257     mpz_fdiv_r(x_2P,x_2P,n);
258     mpz_mul_si(temp2,x_P_copy,2);
259     mpz_sub(x_2P,x_2P,temp2);
260     mpz_fdiv_r(x_2P,x_2P,n);
```

```
261     mpz_sub(temp2,x_P_copy,x_2P);
262     mpz_mul(y_2P,temp1,temp2);
263     mpz_sub(y_2P,y_2P,y_P_copy);
264     mpz_fdiv_r(y_2P,y_2P,n);
265 
266     mpz_clears(y_P2_inv,temp1,temp2,x_P_copy,y_P_copy,NULL);
267     return 1;
268 }
269 
270 // 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
271 // 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
272 // proprio d di n, e restituisco 0.
273 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) {
274     mpz_t denom,temp1;
275     mpz_inits(denom,temp1,NULL);
276     mpz_sub(temp1,x_Q,x_P);
277     inv_mod_mcd(d,denom,temp1,n);
278     if (mpz_cmp(d,n)==0) { // fallimento: restituisco -1
279         mpz_clears(denom,temp1,NULL);
280         return -1;
281     }
282     if (mpz_cmp_si(d,1)!=0) { // successo: restituisco 0
283         mpz_clears(denom,temp1,NULL);
284         return 0;
285     }
286     // calcolo P+Q: vedi Koblitz per le formule
287     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
288     mpz_inits(temp2,x_P_copy,y_P_copy,x_Q_copy,y_Q_copy,NULL);
289     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);
290     mpz_sub(temp1,y_Q_copy,y_P_copy);
291     mpz_mul(temp1,temp1,denom);
292     mpz_fdiv_r(temp1,temp1,n);
293     mpz_mul(x_sum,temp1,temp1);
294     mpz_sub(x_sum,x_sum,x_P_copy);
295     mpz_sub(x_sum,x_sum,x_Q_copy);
296     mpz_fdiv_r(x_sum,x_sum,n);
297     mpz_sub(temp2,x_P_copy,x_sum);
298     mpz_mul(y_sum,temp1,temp2);
299     mpz_sub(y_sum,y_sum,y_P_copy);
300     mpz_fdiv_r(y_sum,y_sum,n);
301 
302     mpz_clears(denom,temp1,temp2,x_P_copy,y_P_copy,x_Q_copy,y_Q_copy,NULL);
303     return 1;
304 }
305 
306 // tenta di calcolare P-->kP, con k>2 intero, utilizzando raddoppi ripetuti (
        analogo a esponenziazione binaria). Nel farlo deve utilizzare le formule
307 // 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
308 // restituisco 0; altrimenti porto a termine il calcolo e restituisco 1.
309 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) {
310     int outcome;
311     mpz_t temp1,temp2;
312     mpz_inits(temp1,temp2,NULL);
313     mpz_set(temp1,x_P); mpz_set(temp2,y_P); // temp conterranno man mano le
            coordinate dei raddoppi successivi di P
314     mpz_set_si(x_kP,0); mpz_set_si(y_kP,0);
315 
316     while (k>0) {
317         if (k&1==1) { // controllo che l'ultimo bit di k sia un 1; se lo è sommo,
                altrimenti lascio così
318 
319             outcome=sum_elliptic(d,x_kP,y_kP,x_kP,y_kP,temp1,temp2,n);
320             if (outcome==-1) {
```

```
321                 mpz_clears(temp1,temp2,NULL);
322                 return -1; // somma non riuscita: devo cambiare curva
323             }
324             if (outcome==0) {
325                 mpz_clears(temp1,temp2,NULL);
326                 return 0; // successo: ho trovato il divisore d
327             }
328             // altrimenti ho portato a termine la somma con successo, e proseguo i
                    calcoli
329         }
330         // aggiorno il raddoppio
331         outcome=double_elliptic(d,temp1,temp2,temp1,temp2,a,n);
332         if (outcome==-1) {
333             mpz_clears(temp1,temp2,NULL);
334             return -1; // raddoppio non riuscito: devo cambiare curva
335         }
336         if (outcome==0) {
337             mpz_clears(temp1,temp2,NULL);
338             return 0; // successo: ho trovato il divisore d
339         }
340         // altrimenti ho portato a termine il raddoppio con successo, e proseguo i
                calcoli
341 
342         k>>=1; // shifto di 1 i bit di k
343     }
344     // se esco dal ciclo ho portato a termine la moltiplicazione P-->kP con
            successo
345 
346     mpz_clears(temp1,temp2,NULL);
347     return 1;
348 }
349 
350 // calcola in x la stima di Hasse per n: hasse(n)= (radice_quarta(n) + 1)^2
351 void hasse (mpz_t x, mpz_t n) {
352     mpz_root(x,n,4);
353     mpz_add_ui(x,x,2); // sommo 2 poiché mpz_root restituisce la radice troncata
            alla parte intera
354     mpz_mul(x,x,x);
355 
356     return;
357 }
358 
359 // calcola gli esponenti a cui deve essere elevato ciascun primo: exp=parte intera
        di log_(base p) (hasse_n)
360 void create_exp_primes (unsigned int* exponents, int* primes, mpz_t hasse_n) {
361     int i;
362     for (i=0; i<NUM_PRIMES; i++) {
363         // trucco per calcolare il logaritmo: mpz_get_str restituisce una stringa
                contenente la rappresentazione di hasse_n in base
364         // primes[i]; per avere la parte intera del logaritmo base p basta quindi
                calcolare la lunghezza della stringa meno 1
365         exponents[i]=strlen(mpz_get_str(NULL,primes[i],hasse_n))-1;
366     }
367 
368     return;
369 }
370 
371 // 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
372 void calcola_b (mpz_t b, mpz_t x, mpz_t y, mpz_t a, mpz_t n) {
373     mpz_t temp;
374     mpz_init(temp);
375     mpz_mul(b,y,y);
376     mpz_fdiv_r(b,b,n);
377     exp_mod_ui(temp,x,3,n);
378     mpz_sub(b,b,temp);
379     mpz_mul(temp,a,x);
380     mpz_sub(b,b,temp);
381     mpz_fdiv_r(b,b,n);
382 
```

```
383     mpz_clear(temp);
384     return;
385 }
386
387 // calcola il discriminante della curva ellittica: delta= 27*a^3 +4*b^2 mod n
388 void calcola_delta (mpz_t delta, mpz_t a, mpz_t b, mpz_t n) {
389     mpz_t temp;
390     mpz_init(temp);
391     mpz_mul_ui(delta,a,3);
392     exp_mod_ui(delta,delta,3,n);
393     mpz_mul_ui(temp,b,2);
394     mpz_mul(temp,temp,temp);
395     mpz_add(delta,delta,temp);
396     mpz_fdiv_r(delta,delta,n);
397
398     mpz_clear(temp);
399     return;
400 }
```

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

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4  #pragma warning(disable:4146)
5  #include <gmp.h>
6
7  // Struttura per memorizzare un baby step: (valore, esponente)
8  typedef struct {
9      mpz_t value;            // memorizza a^j mod m
10     unsigned long exponent; // esponente j
11 } Pair;
12
13 // Funzione di confronto per qsort e bsearch (ordinamento in base a value)
14 int cmp_pair(const void* a, const void* b) {
15     const Pair* pa = (const Pair*)a;
16     const Pair* pb = (const Pair*)b;
17     return mpz_cmp(pa->value, pb->value);
18 }
19
20 /*
21  * Funzione babyStepGiantStep:
22  *  - result: (mpz_t) verra' impostato con la soluzione x, se trovata.
23  *  - a, b, m: valori in mpz_t che definiscono l'equazione a^x = b mod m.
24  *
25  * Restituisce 1 se la soluzione e' trovata, 0 altrimenti.
26  */
27 int babyStepGiantStep(mpz_t result, const mpz_t a, const mpz_t b, const mpz_t m) {
28     // Calcoliamo order = m - 1 (nel caso in cui m sia primo)
29     mpz_t order;
30     mpz_init(order);
```

```
    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;
```

```
            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

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

```
5
6  // Funzione per calcolare il MCD con algoritmo di Euclide
7  // usa la libreria GMP (mpz_gcd)
8  // Algoritmo di Euclide esteso per trovare l'inverso modulo
9  void mod_inverse(mpz_t result, mpz_t a, mpz_t m) {
10     mpz_invert(result, a, m);
11 }
12
13 // Esponenziazione modulare binaria
14 // calcola base^exp % mod, e utilizza la funzione GMP mpz_pow
15 void mod_exp(mpz_t result, mpz_t base, mpz_t exp, mpz_t mod) {
16     mpz_t b, e;
17     mpz_init_set(b, base);
18     mpz_init_set(e, exp);
19     mpz_set_ui(result, 1);
20
21     while (mpz_cmp_ui(e, 0) > 0) {
22         if (mpz_odd_p(e))
23             mpz_mul(result, result, b), mpz_mod(result, result, mod);
24         mpz_fdiv_q_2exp(e, e, 1);
25         mpz_mul(b, b, b), mpz_mod(b, b, mod);
26     }
27
28     mpz_clear(b);
29     mpz_clear(e);
30 }
31
32 // Algoritmo di Baby-Step Giant-Step per trovare il logaritmo discreto modulo un
       primo q
33 // L'algoritmo è una tecnica di ricerca per trovare x tale che: g^x = h (mod p)
34 // dove g è la base, h è il valore di cui vogliamo trovare il logaritmo, e p è un
       modulo primo.
35 // L'algoritmo divide il problema in due parti:
36 // 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).
37 // La giant-step calcola successivamente i valori h * g^(-i*m) % p e cerca se uno
       di questi è nella tabella.
38 // Se trova una corrispondenza, restituisce la soluzione come la somma degli indici
        i e j. Se non trova nulla, restituisce 0.
39 void baby_giant_step(mpz_t result, mpz_t g, mpz_t h, mpz_t p) {
40     mpz_t m, val, g_m, inv_g_m, temp;
41     mpz_init(m);
42     mpz_sqrt(m, p);
43     mpz_add_ui(m, m, 1);
44
45     mpz_init_set_ui(val, 1);
46     mpz_init(temp);
47
48     // Creazione della tabella dei baby steps
49     size_t m_ui = mpz_get_ui(m);
50     mpz_t *table = malloc(m_ui * sizeof(mpz_t));
51     for (size_t i = 0; i < m_ui; i++) {
52         mpz_init_set(table[i], val);
53         mpz_mul(val, val, g);
54         mpz_mod(val, val, p);
55     }
56
57     // Calcola g^(-m) mod p
58     mpz_init(g_m);
59     mpz_invert(g_m, g, p);
60     mpz_init(inv_g_m);
61     mod_exp(inv_g_m, g_m, m, p);
62
63     mpz_set(val, h);
64     for (size_t i = 0; i < m_ui; i++) {
65         for (size_t j = 0; j < m_ui; j++) {
66             if (mpz_cmp(table[j], val) == 0) {
67                 mpz_set_ui(result, i * m_ui + j);
68                 goto cleanup;
69             }
```

```
70          }
71          mpz_mul(val, val, inv_g_m);
72          mpz_mod(val, val, p);
73      }
74      mpz_set_ui(result, 0);
75
76  cleanup:
77      for (size_t i = 0; i < m_ui; i++) {
78          mpz_clear(table[i]);
79      }
80      free(table);
81      mpz_clears(m, val, g_m, inv_g_m, temp, NULL);
82  }
83
84  // Risoluzione del logaritmo discreto modulo q^e usando il metodo di Pohlig-Hellman
85  // approccio efficiente quando il modulo è fattorizzato come un prodotto di potenze
        di primi piccoli
86  // Questa funzione sfrutta la fattorizzazione del modulo p come un prodotto di
        potenze di primi.
87  // La funzione esegue i seguenti passi:
88  // Calcola q^e (dove q è un primo e e è l'esponente corrispondente alla potenza di
        q).
89  // Calcola i valori di g_i e h_i come potenze di g e h modulo p, utilizzando q^e.
90  // Risolve il logaritmo discreto di h_i rispetto a g_i modulo q^e usando l'
        algoritmo di Baby-Step Giant-Step.
91  // Restituisce il risultato parziale del logaritmo discreto modulo q^e.
92  // Funzione per Pohlig-Hellman per un fattore q^e
93  void pohlig_hellman(mpz_t result, mpz_t g, mpz_t h, mpz_t p, mpz_t q, int exp) {
94      mpz_t q_e, exponent, g_i, h_i;
95      mpz_inits(q_e, exponent, g_i, h_i, NULL);
96
97      // q_e = q^exp
98      mpz_pow_ui(q_e, q, exp);
99      // Calcola l'esponente: (p-1) / q^exp
100     mpz_sub_ui(exponent, p, 1);
101     mpz_divexact(exponent, exponent, q_e);
102     // g_i = g^( (p-1)/q^exp ) mod p, h_i = h^( (p-1)/q^exp ) mod p
103     mod_exp(g_i, g, exponent, p);
104     mod_exp(h_i, h, exponent, p);
105
106     baby_giant_step(result, g_i, h_i, p);
107
108     mpz_clears(q_e, exponent, g_i, h_i, NULL);
109 }
110
111 // Teorema Cinese del Resto per combinare i risultati modulo q_i^e_i
112 // questa funzione combina i risultati del logaritmo discreto calcolati per moduli
        diversi, ottenuti tramite il metodo di Pohlig-Hellman.
113 // Il teorema permette di risolvere un sistema di congruenze, restituendo un
        risultato finale modulo M, dove M è il prodotto di tutti i moduli.
114 // La formula del CRT è la seguente:
115 // x = sum_i{xi*Mi·inv(Mi)) (mod M)
116 // 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.
117 void chinese_remainder(mpz_t result, mpz_t *x, mpz_t *m, size_t len) {
118     mpz_t M, Mi, inv, sum;
119     mpz_init_set_ui(M, 1);
120     for (size_t i = 0; i < len; i++) {
121         mpz_mul(M, M, m[i]);
122     }
123
124     mpz_init_set_ui(sum, 0);
125     for (size_t i = 0; i < len; i++) {
126         mpz_init(Mi);
127         mpz_divexact(Mi, M, m[i]);
128         mpz_init(inv);
129         mod_inverse(inv, Mi, m[i]);
130         mpz_mul(Mi, Mi, inv);
131         mpz_mul(Mi, Mi, x[i]);
132         mpz_add(sum, sum, Mi);
```

```
133         mpz_clear(Mi);
134         mpz_clear(inv);
135     }
136     mpz_mod(result, sum, M);
137 
138     mpz_clears(M, sum, NULL);
139 }
140 
141 // Algoritmo di Pohlig-Hellman completo
142 // Funzione principale per il logaritmo discreto
143 // utilizza tutte le precedenti per risolvere il logaritmo discreto in modo efficiente.
144 // 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.
145 // Combina i risultati ottenuti usando il Teorema Cinese del Resto.
146 void pohlig_hellman_algorithm(mpz_t result, mpz_t g, mpz_t h, mpz_t p, mpz_t *q, int *e, size_t len) {
147     mpz_t *x = malloc(len * sizeof(mpz_t));
148     mpz_t *moduli = malloc(len * sizeof(mpz_t));
149 
150     for (size_t i = 0; i < len; i++) {
151         mpz_init(moduli[i]);
152         mpz_pow_ui(moduli[i], q[i], e[i]);  // moduli[i] = q[i]^(e[i])
153         mpz_init(x[i]);
154         pohlig_hellman(x[i], g, h, p, q[i], e[i]);
155     }
156 
157     chinese_remainder(result, x, moduli, len);
158 
159     for (size_t i = 0; i < len; i++) {
160         mpz_clear(x[i]);
161         mpz_clear(moduli[i]);
162     }
163     free(x);
164     free(moduli);
165 }
166 
167 // Main
168 // Legge i valori di g (base), h (target), e p (modulo).
169 // Fa inserire all'utente la fattorizzazione di p, con una lista di primi q con i loro esponenti e .
170 // Chiama pohligHellmanAlgorithm() per calcolare il logaritmo discreto.
171 // Stampa il risultato finale.
172 int main() {
173     mpz_t g, h, p, result;
174     mpz_inits(g, h, p, result, NULL);
175 
176     printf("Inserisci base g, valore h e modulo p (p primo):\n");
177     gmp_scanf("%Zd %Zd %Zd", g, h, p);
178 
179     // Calcoliamo l'ordine del gruppo: p - 1
180     mpz_t order;
181     mpz_init(order);
182     mpz_sub_ui(order, p, 1);
183 
184     int num_factors;
185     printf("Inserisci il numero di fattori distinti nella fattorizzazione di (p-1): ");
186     scanf("%d", &num_factors);
187 
188     // Allochiamo dinamicamente gli array per i fattori e gli esponenti
189     mpz_t *q = malloc(num_factors * sizeof(mpz_t));
190     int *exp_array = malloc(num_factors * sizeof(int));
191 
192     for (int i = 0; i < num_factors; i++) {
193         mpz_init(q[i]);
194         printf("Fattore primo #%d: ", i + 1);
195         gmp_scanf("%Zd", q[i]);
196         printf("Esponente per questo fattore: ");
197         scanf("%d", &exp_array[i]);
```

```
    }

    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

```
#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

```

```
40          mpz_sub_ui(p1, p, 1); // p1 = p - 1;
41          mpz_set_si(it, 0);
42
43          bool test=false;
44
45          while (mpz_cmp(it, p)<0) {
46                  funz(gamma, a, b, p, p1, g, h);
47                  funz(lambda, alpha, beta, p, p1, g, h);
48                  funz(lambda, alpha, beta, p, p1, g, h); // si muove al doppio della
                        velocità
49
50                  if (mpz_cmp(gamma, lambda)==0) {
51                          mpz_sub(temp_b, beta, b);
52                          mpz_sub(temp_a, a, alpha);
53                          mcd_euclide(d, temp_b, p1);
54                          if (mpz_cmp_si(d, 1)==0) {
55                                  inv_mod(invb, temp_b, p1); // calcolo l'inverso di
                                        temp_b mod p1
56                                  mpz_mul(x, invb, temp_a);
57                                  mpz_mod(x, x, p1);
58                                  break;
59                          } else {
60                                  mpz_fdiv_q(pd, p1, d); // calcolo p2 = p1 / d
61                                  mpz_fdiv_q(c, temp_b, d);
62                                  mpz_fdiv_q(ad, temp_a, d);
63                                  inv_mod(invc, c, pd); // calcolo l'inverso di c mod
                                        p2
64                                  mpz_mul(x, invc, ad);
65                                  mpz_mod(x, x, pd); // unica soluzione mod p1/d (
                                        ossia pd)
66
67                                  // studio a mano le d soluzioni mod p1
68                                  mpz_t i;
69                                  mpz_inits(i, NULL);
70                                  mpz_set_si(i, 0);
71                                  while (mpz_cmp(i, d)<0) {
72                                          if (mpz_cmp_si(i, 0)!=0) {
73                                                  mpz_add(x, x, d); // x = x + d
74                                          }
75                                          mpz_powm(temp, g, x, p);
76                                          if (mpz_cmp(temp, h)==0) {
77                                                  test=true;
78                                                  break;
79                                          }
80                                          mpz_add_ui(i, i, 1);
81                                  }
82                                  mpz_clears(i, NULL);
83                          }
84                  }
85                  if (test==true) {
86                          break;
87                  }
88
89                  mpz_add_ui(it, it, 1);
90          }
91
92          mpz_clears(gamma, a, b, lambda, alpha, beta, p1, temp_a, temp_b, invb, d, c
                , invc, pd, ad, temp, it, NULL);
93          return;
94  }
95
96  void funz (mpz_t x, mpz_t a, mpz_t b, mpz_t p, mpz_t p1, mpz_t g, mpz_t h) {
97          mpz_t temp;
98          mpz_inits(temp, NULL);
99          // suddividiamo il campo in 3 sottoinsiemi di uguale cardinalità con le
                classi di resto mod 3
100         mpz_fdiv_r_ui(temp, x, 3); // temp = x % 3
101
102         if (mpz_cmp_si(temp, 0)==0) {
103                 mpz_mul(x, x, x); // x = x * x
```

```
104                     mpz_mod(x, x, p); // riporto x mod p
105                     mpz_mul_ui(a, a, 2); // a = a * 2
106                     mpz_mod(a, a, p1); // riporto a mod p-1
107                     mpz_mul_ui(b, b, 2); // b = b * 2
108                     mpz_mod(b, b, p1); // riporto b mod p-1
109             } else if (mpz_cmp_si(temp, 1)==0) {
110                     mpz_mul(x, x, g); // x = x * g
111                     mpz_mod(x, x, p); // riporto x mod p
112                     mpz_add_ui(a, a, 1); // a = a + 1
113                     mpz_mod(a, a, p1); // riporto a mod p-1
114                     // b resta invariato
115             } else {
116                     mpz_mul(x, x, h); // x = x * g
117                     mpz_mod(x, x, p); // riporto x mod p
118                     // a resta invariato;
119                     mpz_add_ui(b, b, 1); // a = a + 1
120                     mpz_mod(b, b, p1); // riporto a mod p-1
121             }
122
123             mpz_clears(temp, NULL);
124             return;
125     }
```

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);
```

```
59      // inizializzo alpha=x^2*a^-1
60      exp_mod(alpha,a_modp,d,p);
61
62      h--;
63      mpz_init_set_si(exp,1);
64      mpz_mul_2exp(exp,exp,h);
65
66  /*  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
67      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).
68      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
69      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
70      di 1, e aggiornare alpha di conseguenza*/
71
72      while (h>0) {
73          // aggiorno h e exp
74          h--;
75          mpz_fdiv_q_2exp(exp,exp,1);
76          exp_mod(temp,alpha,exp,p);
77          // testo su alpha
78          if (mpz_cmp_si(temp,1)!=0) { // test fallito: aggiorno x e alpha
79              mpz_mul(x,x,beta);
80              mpz_fdiv_r(x,x,p);
81
82              mpz_mul(alpha,x,x);
83              mpz_mul(alpha,alpha,a_inv);
84              mpz_fdiv_r(alpha,alpha,p);
85          }
86
87          mpz_mul(beta,beta,beta); // aggiorno beta che diventa il suo quadrato
88          mpz_fdiv_r(beta,beta,p);
89      }
90      // quando esco h=0 ossia alpha=x^2*a^-1 è radice 1-esima di 1, quindi x radice
            di a
91
92      mpz_clears(a_modp,d,alpha,beta,a_inv,exp,temp,NULL);
93      gmp_randclear(randstate);
94      return 1;
95  }
```

3.2 Sistemi di interi

Listing 26: Sis interi.c

```
1   #include "..\Include\Matrix.h"
2
3   //risolvo un sistema di interi con n incognite e smith
4
5   int main() {
6
7           int n = 0;
8           printf("Quante incognite ha il sistema? n = ");
9           scanf("%d", &n);
10
11          //richiedo i dati per la matrice e il termine noto
12          int** matrix = NULL;
13          matrix = input_null(matrix, n, n);
```

```
14
15          for (int i = 0; i < n; i++) {
16                  for (int j = 0; j < n; j++) {
17                          printf("Inserisci l'elemento %d %d della matrice: ", i, j);
18                          scanf("%d", &matrix[i][j]);
19                  }
20          }
21
22          int** B = NULL;
23          B = input_null(B, n, 1);
24          for (int i = 0; i < n; i++) {
25                  printf("Inserisci il termine noto %d: ", i);
26                  scanf("%d", &B[i][0]);
27          }
28
29          //calcolo la forma normale di Smith della matrice
30          int** S = NULL;
31          S = input_id(S, n);
32          int** S_inv = NULL;
33          S_inv = input_id(S_inv, n);
34          int** T = NULL;
35          T = input_id(T, n);
36          int** T_inv = NULL;
37          T_inv = input_id(T_inv, n);
38          int** D = NULL;
39          D = input_null(D, n, n);
40
41          for (int i = 0; i < n; i++) {
42                  for (int j = 0; j < n; j++) {
43                          D[i][j] = matrix[i][j];
44                  }
45          }
46
47          //risolvo Ax = B con la forma normale di Smith
48
49          SmithNormalForm5mat(D, S, T, S_inv, T_inv, n, n);
50
51          //SDT x = B
52
53          //Y = DT^-1 B
54          int** Y = NULL;
55          Y = mul_matrix(S, n, n, B, n, 1);
56
57
58          for (int i = 0; i < n; i++) {
59                  for (int j = 0; j < 1; j++) {
60                          if (D[i][i] == 0) {
61                                  Y[i][j] = 0;
62                          }
63                          else if (Y[i][j] % D[i][i] != 0) {
64                                  printf("Il sistema non ha soluzione negli interi\n"
                                        );
65                                  return 1;
66                          }
67                          else {
68                                  Y[i][j] = Y[i][j] / D[i][i];
69                          }
70                  }
71          }
72
73          print_matrix(Y, n, 1);
74
75          //X = TY
76          int** X = NULL;
77          X = mul_matrix(T, n, n, Y, n, 1);
78
79          printf("La soluzione del sistema con matrice A :\n");
80          print_matrix(matrix, n, n);
81          printf("e termine noto B : \n");
82          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];
```

```
105                 }
106             }
107 
108             // Elimina tutti gli altri elementi nella colonna i
109             for (int k = 0; k < size; k++) {
110                 if (k != i && A[k][i] != 0) {
111                     int factor = A[k][i];  // in una matrice unimodulare dovrebbe
                            essere più o meno 1
112                     for (int j = 0; j < size; j++) {
113                         A[k][j] -= factor * A[i][j];
114                         inv[k][j] -= factor * inv[i][j];
115                     }
116                 }
117             }
118         }
119 
120         free(A);
121         return inv;
122     }
123 
124     //Funzioni per calcolare l'eliminazione di gauss
125     void gauss(float** matrice, int n) {
126         for (int i = 0; i < n; i++) {
127             // Pivot
128             if (matrice[i][i] == 0) {
129                 for (int k = i + 1; k < n; k++) {
130                     if (matrice[k][i] != 0) {
131                         // Scambia righe
132                         for (int j = 0; j < n; j++) {
133                             float temp = matrice[i][j];
134                             matrice[i][j] = matrice[k][j];
135                             matrice[k][j] = temp;
136                         }
137                         break;
138                     }
139                 }
140             }
141 
142             // Normalizza la riga pivot
143             float pivot = matrice[i][i];
144             for (int j = 0; j < n; j++) {
145                 matrice[i][j] /= pivot;
146             }
147 
148             // Eliminazione verso il basso
149             for (int k = i + 1; k < n; k++) {
150                 float coeff = matrice[k][i];
151                 for (int j = 0; j < n; j++) {
152                     matrice[k][j] -= coeff * matrice[i][j];
153                 }
154             }
155         }
156     }
157 
158     //Funzione per calcolare Gauss di una matrice rettangolare senza scambiare le
            righe quindi non viene con le righe in ordine
159         void gauss_rectangular(int** matrice, int row, int col) {
160                 int r = 0;
161                 for (int i = 0; i < col; i++) {
162 
163                         // Pivot
164             if (matrice[i][i] == 0) {
165                 for (int k = i + 1; k < row; k++) {
166                     if (matrice[k][i] != 0) {
167                         printf("matrice[%d][%d] = %d\n", k, i, matrice[k][i]);
168                         // Non scambio le righe perchè mi serve sapere quali sono
                                le righe non nulle
169                         r = k;
170                         break;
171                     }
```

```
                    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);
```

```
241                         free(T);
242                         return B;
243                 }
244         B = input_null(B, c, c - rk);
245                 for (int i = 0; i < c; i++) {
246                         for (int j = 0; j < c - rk; j++) {
247                                 B[i][j] = T[i][j + rk];
248                         }
249                 }
250
251                 //print_matrix(B, c, c - rk);
252
253                 free(D);
254                 free(S);
255                 free(T);
256                 *n = c - rk;
257                 return B;
258     }
259
260
261         //Funzione per concatenare due matrici con lo stesso numero di righe
262         int** link2matrix_same_row(int** matrix1, int row1, int col1, int** matrix2
                , int row2, int col2) {
263                 int** result = NULL;
264                 if (row1 != row2) {
265                         printf("Errore: Le matrici non hanno lo stesso numero di
                            righe.\n");
266                         return NULL;
267                 }
268                 result = input_null(result, row1, col1 + col2);
269                 for (int i = 0; i < row1; i++) {
270                         for (int j = 0; j < col1; j++) {
271                                 result[i][j] = matrix1[i][j];
272                         }
273                         for (int j = 0; j < col2; j++) {
274                                 result[i][j + col1] = matrix2[i][j];
275                         }
276                 }
277                 return result;
278         }
279
280         //Funzione per calcolare il rango di una matrice tramite smith
281     //portiamo la amtrice in forma diagonale e calcoliamo il rango della matrice
            diagonale molto più veloce
282         int rank_matrix(int** M, int r, int c) {
283         int** D = NULL;
284         int** S = NULL;
285         int** T = NULL;
286
287         D = input_null(D, r, c);
288         S = input_id(S, r);
289         T = input_id(T, c);
290
291         for (int i = 0; i < r; i++) {
292             for (int j = 0; j < c; j++) {
293                 D[i][j] = M[i][j];
294             }
295         }
296
297
298
299         SmithNormalForm(D, S, T, r, c);
300         int rk = rank_matrix_diag(D, r, c);
301
302         free(D);
303                 free(S);
304                 free(T);
305
306                 return rk;
307         }
```

```
308
309     char* matrix_to_json(int** matrix, int n) {
310         // Calcoliamo una dimensione massima per il buffer.
311         // Per ogni numero, assumiamo al massimo 12 caratteri (inclusi segno, cifra
                e separatore).
312         int buffer_size = n * (n * 12 + 2) + 2;
313         char* buffer = (char*)malloc(buffer_size);
314         if (!buffer) {
315             perror("Errore nell'allocazione della memoria");
316             exit(EXIT_FAILURE);
317         }
318         int pos = 0;
319         pos += snprintf(buffer + pos, buffer_size - pos, "[");
320         for (int i = 0; i < n; i++) {
321             pos += snprintf(buffer + pos, buffer_size - pos, "[");
322             for (int j = 0; j < n; j++) {
323                 pos += snprintf(buffer + pos, buffer_size - pos, "%d", matrix[i][j
                        ]);
324                 if (j < n - 1) {
325                     pos += snprintf(buffer + pos, buffer_size - pos, ",");
326                 }
327             }
328             pos += snprintf(buffer + pos, buffer_size - pos, "]");
329             if (i < n - 1) {
330                 pos += snprintf(buffer + pos, buffer_size - pos, ",");
331             }
332         }
333         pos += snprintf(buffer + pos, buffer_size - pos, "]");
334         return buffer;
335     }
336
337 #endif
```

Listing 29: Fattorizzazione.h

```
1  #include <stdlib.h>
2  #include <gmp.h>
3
4  // definizione della struttura dei fattori: l'obiettivo e' costruire una lista di
       fattori in modo da poter scrivere un intero
5  // come prodotto di primi (in ordine), ciascuno con la rispettiva potenza.
6  typedef struct Factor {
7      mpz_t prime;
8      unsigned int exponent;
9      struct Factor* next;
10 } Factor;
11
12 Factor* add_factor(Factor*, mpz_t, unsigned int);
13 Factor* cons(Factor*, mpz_t, unsigned int);
14 void print_factors(Factor*);
15
16 // aggiunge in testa alla lista il primo p elevato alla potenza exp
17 Factor* cons(Factor* factor, mpz_t p, unsigned int exp) {
18     Factor* new_factor = (Factor*)malloc(sizeof(Factor));
19     mpz_init_set(new_factor->prime,p);
20     new_factor->exponent=exp;
21     new_factor->next=factor;
22     return new_factor;
23 }
24
25 // aggiunge alla lista di fattori "factor", in modo ordinato, il primo p elevato
       alla potenza exp
26 Factor* add_factor(Factor* factor, mpz_t p, unsigned int exp) {
27     // caso in cui sono arrivato a fine lista, oppure ho trovato il primo numero
           primo presente nella lista maggiore di p: aggiungo un fattore
28     if (factor==NULL || mpz_cmp(factor->prime,p)>0) {
29         return cons(factor,p,exp);
```

```
30      }
31
32      // caso in cui ho trovato nella lista un primo identico a p: sommo l'esponente
33      if (mpz_cmp(factor->prime,p)==0) {
34          factor->exponent=factor->exponent+exp;
35          return factor;
36      }
37
38      // proseguo la ricerca in modo ricorsivo
39      factor->next=add_factor(factor->next,p,exp);
40      // riaggancio i puntatori al ritorno della ricorsione
41      return factor;
42  }
43
44  // stampa la lista dei fattori
45  void print_factors(Factor* factor) {
46      if (factor==NULL) return;
47
48      gmp_printf("%Zd^(%u)",factor->prime,factor->exponent);
49      factor=factor->next;
50      while (factor!=NULL) {
51          gmp_printf(" * %Zd^(%u)",factor->prime,factor->exponent);
52          factor=factor->next;
53      }
54
55      return;
56  }
```