

Istituzioni di Algebra e geometria

Andrea Agostini 1996124

Simone Bonanni 1992505

Giacomo Gneri 2025964

Marta Graziano 2024185

Elena Sbordonì 2000180

Febbraio 2025

Indice

1	Geometria: analisi dati	3
1.1	Complessi simpliciali	3
1.2	Complessi di catene e omologia simpliciale	13
1.3	Forma normale di Smith	25
1.4	Categorie, funtori e moduli di persistenza	33
2	Algebra: crittografia	44
2.1	Ideali su \mathbb{Z} , MCD e identità di Bézout	44
2.2	Teorema cinese dei resti	53
2.3	Test di primalità	55
2.3.1	Test di Fermat	55
2.3.2	Test di Eulero	60
2.3.3	Test di Solovay-Strassen	62
2.3.4	Test di Miller-Rabin	63
2.4	Fattorizzazione di numeri composti	66
2.4.1	Metodo rho di Pollard	66
2.4.2	Basi di primi di Pomerance	67
2.4.3	Metodo $p - 1$ di Pollard	68
2.4.4	Algoritmo di Lenstra	69
2.5	Logaritmo discreto	74
2.6	Sistema a chiave pubblica RSA	82
3	Codici ausiliari	84
3.1	Radici in \mathbb{Z}_p	84
3.2	Sistemi di interi	85
3.3	Frazioni Continue	87
3.4	Funzioni ausiliarie	88

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. $\tau \in S \Rightarrow \sigma \in S \quad \forall \sigma \subset \tau$
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 semplice 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

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <stdbool.h>
4
5 // Definizione della struttura "simplex"
6 typedef struct simplex {
7     int* vertices;           // Array dinamico contenente i vertici (ordinato)
8     int position;           // Numero di vertici (dimensione + 1)
9     struct simplex* next;    // Puntatore al semplice successivo nella lista
10    concatenata
11 } simplex;
12
13 // Definizione della struttura "SimplicialComplex"
14 typedef struct {
15     simplex* simplices;     // Puntatore alla lista concatenata di simplessi
16     int size;               // Numero di simplessi presenti
17 } SimplicialComplex;
18
19 // Funzione di confronto per qsort (ordinamento crescente)
20 int cmpInt(const void *a, const void *b) {
21     int int_a = *(const int *)a;
22     int int_b = *(const int *)b;
23     return int_a - int_b;
24 }
25
26 // Funzione che verifica se il semplice "sub" è un sottoinsieme del semplice "sup"
27 // Entrambi gli array di vertici devono essere ordinati in ordine crescente.
28 bool isSubset(simplex* sub, simplex* sup) {
29     int i = 0, j = 0;
30     while (i < sub->position && j < sup->position) {
31         if (sub->vertices[i] == sup->vertices[j]) {
32             i++;
33             j++;
34         }
35     }
36     return i == sub->position;
```

```

32         j++;
33     } else if (sub->vertices[i] > sup->vertices[j]) {
34         j++;
35     } else { // sub->vertices[i] < sup->vertices[j]
36         return false;
37     }
38 }
39 return (i == sub->position);
40 }
41
42 // Funzione per aggiungere un semplice alla struttura SimplicialComplex
43 void addSimplex(SimplicialComplex* complex, int* vertices, int numVertices) {
44     simplex* newNode = (simplex*)malloc(sizeof(simplex));
45     if(newNode == NULL) {
46         fprintf(stderr, "Errore di allocazione della memoria.\n");
47         exit(1);
48     }
49     newNode->vertices = vertices;
50     newNode->position = numVertices;
51     newNode->next = complex->simplices;
52     complex->simplices = newNode;
53     complex->size++;
54 }
55
56 int main(void) {
57     int k;
58     printf("Inserisci il numero massimo di dimensioni (k): ");
59     if (scanf("%d", &k) != 1) {
60         fprintf(stderr, "Input non valido.\n");
61         return 1;
62     }
63
64     // Allocazione dinamica di un array di SimplicialComplex per dimensioni da 0 a
65     // k.
66     SimplicialComplex* complexes = (SimplicialComplex*)malloc((k + 1) * sizeof(
67     SimplicialComplex));
68     if (complexes == NULL) {
69         fprintf(stderr, "Errore di allocazione della memoria.\n");
70         return 1;
71     }
72
73     int i, dim, s, j, supDim;
74     // Inizializzazione dei complessi
75     for (i = 0; i <= k; i++) {
76         complexes[i].simplices = NULL;
77         complexes[i].size = 0;
78     }
79
80     // Costruzione automatica degli 0-simplessi
81     int numZeroSimplices;
82     printf("Inserisci il numero di 0-simplessi: ");
83     if (scanf("%d", &numZeroSimplices) != 1) {
84         fprintf(stderr, "Input non valido.\n");
85         free(complexes);
86         return 1;
87     }
88
89     for (i = 0; i < numZeroSimplices; i++) {
90         int* vertex = (int*)malloc(sizeof(int));
91         if (vertex == NULL) {
92             fprintf(stderr, "Errore di allocazione della memoria.\n");
93             exit(1);
94         }
95         vertex[0] = i + 1; // Gli 0-simplessi sono rappresentati da un singolo
96         // vertice
97         addSimplex(&complexes[i], vertex, 1);
98     }
99
100     // Input e costruzione dei semplici per dimensioni maggiori di 0.
101     // Per ogni dimensione dim (da 1 a k), ogni semplice ha (dim + 1) elementi.
102     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 simplexso di dimensione inferiore a k, si verifica se è contenuto
127 // in un simplexso di dimensione superiore.
128 for (dim = k - 1; dim >= 0; dim--) {
129     simplex* newList = NULL;
130     int newSize = 0;
131     simplex* current = complexes[dim].simplices;
132     while (current != NULL) {
133         bool isMaximal = true;
134         for (supDim = dim + 1; supDim <= k && isMaximal; supDim++) {
135             simplex* candidate = complexes[supDim].simplices;
136             while (candidate != NULL && isMaximal) {
137                 if (isSubset(current, candidate)) {
138                     isMaximal = false;
139                 }
140                 candidate = candidate->next;
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
158 // Output dei complessi massimali
159 printf("I complessi massimali sono:\n");
160 for (dim = 0; dim <= k; dim++) {
161     if (complexes[dim].simplices != NULL) {
162         printf("%d-simplessi massimali:\n", dim);
163         simplex* cur = complexes[dim].simplices;
164         while (cur != NULL) {
165             printf("{");
166             for (j = 0; j < cur->position; j++) {
167                 if (j > 0) {
168                     printf(", ");
169                 }
170             }
171             printf("}");
172             cur = cur->next;
173         }
174     }
175 }

```

```

168         }
169         printf("%d", cur->vertices[j]);
170     }
171     printf("}\n");
172     cur = cur->next;
173 }
174 }
175 }
176
177 // Deallocazione della memoria utilizzata
178 for (dim = 0; dim <= k; dim++) {
179     simplex* cur = complexes[dim].simplices;
180     while (cur != NULL) {
181         simplex* next = cur->next;
182         free(cur->vertices);
183         free(cur);
184         cur = next;
185     }
186 }
187 free(complexes);
188
189 return 0;
190 }

```

Listing 2: MaxToAll.c

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <stdbool.h>
4
5  typedef struct simplex {
6      int* vertices;           // Array dinamico dei vertici (già ordinato)
7      int position;           // Numero di vertici (dimensione + 1)
8      struct simplex* next;    // Puntatore al semplice successivo (lista
9                               // concatenata)
10 } simplex;
11
12 typedef struct {
13     simplex* simplices;      // Puntatore alla lista concatenata dei semplici
14     int size;                 // Numero di semplici presenti
15 } SimplicialComplex;
16
17 // Funzione di confronto per qsort (ordinamento crescente)
18 int cmpInt(const void *a, const void *b) {
19     int int_a = *(const int*)a;
20     int int_b = *(const int*)b;
21     return int_a - int_b;
22 }
23
24 // Aggiunge un semplice alla struttura SimplicialComplex
25 void addSimplex(SimplicialComplex* complex, int* vertices, int numVertices) {
26     simplex* newNode = (simplex*)malloc(sizeof(simplex));
27     if(newNode == NULL) {
28         fprintf(stderr, "Errore di allocazione della memoria.\n");
29         exit(1);
30     }
31     newNode->vertices = vertices;
32     newNode->position = numVertices;
33     newNode->next = complex->simplices;
34     complex->simplices = newNode;
35     complex->size++;
36 }
37
38 // Confronta due semplici (si assume che abbiano lo stesso numero di vertici)
39 int compareSimplex(const simplex* a, const simplex* b) {
40     int i;
41     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 semplice massimale e li aggiunge
74 // alla struttura dei semplici "allComplexes" in base alla loro dimensione.
75 void generateSubsetsForSimplex(const simplex* maxSimplex, SimplicialComplex*
76     allComplexes) {
77     int n = maxSimplex->position;
78     int total = 1 << n; // 2^n
79     int i, j;
80     for(i = 1; i < total; i++) {
81         int count = 0;
82         for(j = 0; j < n; j++) {
83             if(i & (1 << j))
84                 count++;
85         }
86         int* subset = (int*)malloc(count * sizeof(int));
87         if(subset == NULL) {
88             fprintf(stderr, "Errore di allocazione della memoria.\n");
89             exit(1);
90         }
91         int index = 0;
92         for(j = 0; j < n; j++) {
93             if(i & (1 << j)) {
94                 subset[index] = maxSimplex->vertices[j];
95                 index++;
96             }
97         }
98         // La dimensione del semplice è count-1 (numero di vertici - 1)
99         int subsetDim = count - 1;
100         addSimplex(&allComplexes[subsetDim], subset, count);
101     }
102 }
103
104 // Libera tutta la memoria allocata per un SimplicialComplex
105 void freeSimplicialComplex(SimplicialComplex* complex) {
106     simplex* cur = complex->simplices;
107     simplex* next;
108     while(cur != NULL) {
109         next = cur->next;
110         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 semplici
126     // massimali e uno per tutti quelli generati)
127     SimplicialComplex* maxSimplicialComplexes = (SimplicialComplex*)malloc((k + 1)
128     * sizeof(SimplicialComplex));
129     SimplicialComplex* allSimplicialComplexes = (SimplicialComplex*)malloc((k + 1)
130     * sizeof(SimplicialComplex));
131     if(maxSimplicialComplexes == NULL || allSimplicialComplexes == NULL) {
132         fprintf(stderr, "Errore di allocazione della memoria.\n");
133         exit(1);
134     }
135
136     // Inizializzazione delle strutture per ogni dimensione
137     for(dim = 0; dim <= k; dim++) {
138         maxSimplicialComplexes[dim].simplices = NULL;
139         maxSimplicialComplexes[dim].size = 0;
140         allSimplicialComplexes[dim].simplices = NULL;
141         allSimplicialComplexes[dim].size = 0;
142     }
143
144     // Input dei semplici massimali per ogni dimensione
145     for(dim = 0; dim <= k; dim++) {
146         int numMax;
147         printf("Inserisci il numero di %d-simplici massimali: ", dim);
148         if(scanf("%d", &numMax) != 1) {
149             fprintf(stderr, "Input non valido.\n");
150             exit(1);
151         }
152         for(i = 0; i < numMax; i++) {
153             int m = dim + 1;
154             int* vertices = (int*)malloc(m * sizeof(int));
155             if(vertices == NULL) {
156                 fprintf(stderr, "Errore di allocazione della memoria.\n");
157                 exit(1);
158             }
159             printf("Inserisci gli elementi del %d-simplesso massimale %d: ", dim, i
160             + 1);
161             for(j = 0; j < m; j++) {
162                 if(scanf("%d", &vertices[j]) != 1) {
163                     fprintf(stderr, "Input non valido.\n");
164                     exit(1);
165                 }
166             }
167             qsort(vertices, m, sizeof(int), cmpInt);
168             addSimplex(&maxSimplicialComplexes[dim], vertices, m);
169         }
170     }
171
172     // Per ogni semplice massimale, genera tutti i suoi sottoinsiemi
173     simplex* current;
174     for(dim = 0; dim <= k; dim++) {
175         current = maxSimplicialComplexes[dim].simplices;
176         while(current != NULL) {
177             generateSubsetsForSimplex(current, allSimplicialComplexes);
178             current = current->next;
179         }
180     }

```



```

176     }
177 }
178
179 // Rimuove eventuali duplicati per ciascuna dimensione
180 for(dim = 0; dim <= k; dim++) {
181     removeDuplicates(&allSimplicialComplexes[dim]);
182 }
183
184 // Output di tutti i semplici generati per ogni dimensione
185 printf("Tutti i semplici generati sono:\n");
186 for(dim = 0; dim <= k; dim++) {
187     if(allSimplicialComplexes[dim].simplices != NULL) {
188         printf("%d-simplici generati:\n", dim);
189         current = allSimplicialComplexes[dim].simplices;
190         while(current != NULL) {
191             printf("{");
192             for(i = 0; i < current->position; i++) {
193                 if(i > 0)
194                     printf(", ");
195                 printf("%d", current->vertices[i]);
196             }
197             printf("}\n");
198             current = current->next;
199         }
200     }
201 }
202
203 // Deallocazione della memoria utilizzata
204 for(dim = 0; dim <= k; dim++) {
205     freeSimplicialComplex(&maxSimplicialComplexes[dim]);
206     freeSimplicialComplex(&allSimplicialComplexes[dim]);
207 }
208 free(maxSimplicialComplexes);
209 free(allSimplicialComplexes);
210
211 return 0;
212 }

```

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 semplici.h

```

1  #include "Compl_Simpl.h"
2
3  SimplicialComplex* complex_from_adjacency_matrix_complete (int**, int);
4  SimplicialComplex* complex_from_adjacency_matrix_truncated (int**, int, int);
5  SimplicialComplex* complex_from_1_simplices_complete (SimplicialComplex*, int);
6  SimplicialComplex* complex_from_1_simplices_truncated (SimplicialComplex*, int, int
7  );
8  void add_k_simplices_adjacency_matrix (SimplicialComplex*, int**, int, int);
9
10 int check_neighbor_adjacency_matrix (int**, int*, int, int);
11 int check_neighbor (SimplicialComplex*, int, int);
12
13 // Costruisce il complesso simpliciale completo a partire dagli 1-simplici,
14 // utilizzando la matrice di adiacenza.
15 // I vertici sono nominati da 0 a n-1, come gli indici della matrice.
16 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
            add_k_simplices_adjacency_matrix(complex,M,n,i);
44         }
45         else {
46             complex[i].simplices=NULL;
47             complex[i].size=0;
48         }
49     }
50
51     return complex;
52 }
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
87         esistono degli (i-1)-simplessi
88         add_k_simplices_adjacency_matrix(complex,M,n,i);
89     }
90     else {
91         complex[i].simplices=NULL;
92         complex[i].size=0;
93     }
94 }
95 return complex;
96 }
97
98 // Costruisce il complesso simpliciale completo a partire dagli n vertici e dagli
99 // 1-simplessi.
100 // I vertici verranno rinominati a partire da 0 e con interi consecutivi (se
101 // vogliamo posso scrivere una funzione per farli ritornare ai nomi originali)
102 SimplicialComplex* complex_from_1_simplices_complete (SimplicialComplex* complex,
103 int n) {
104     int i,j;
105     int** M = (int**) malloc(n*sizeof(int*));
106     for (i=0; i<n; i++) M[i] = (int*) malloc(n*sizeof(int));
107     // salvo per comodita' in un array i vertici (per ovviare a problemi di
108     // nomenclatura ed evitare di scorrere sempre la lista)
109     int* vertices = (int*) malloc(n*sizeof(int));
110     Simplex* current=complex[0].simplices;
111     i=0;
112     while (current!=NULL) {
113         vertices[i]=current->vertices[0];
114         i++;
115         current=current->next;
116     }
117
118     // costruisco la matrice di adiacenza
119     for (i=0; i<n; i++) {
120         M[i][i]=1;
121         for (j=i+1; j<n; j++) {
122             // controllo se il vertice i-esimo e il j-esimo sono collegati da un 1-
123             // semplice
124             if (check_neighbor(complex,vertices[i],vertices[j])!=0) {
125                 M[i][j]=1; M[j][i]=1;
126             }
127             else {
128                 M[i][j]=0; M[j][i]=0;
129             }
130         }
131     }
132     return complex_from_adjacency_matrix_complete(M,n);
133 }
134
135 // Costruisce il complesso simpliciale a partire dagli n vertici e dagli 1-
136 // simplessi troncato fino ai k-simplessi (inclusi).
137 // I vertici verranno rinominati a partire da 0 e con interi consecutivi (se
138 // vogliamo posso scrivere una funzione per farli ritornare ai nomi originali)
139 SimplicialComplex* complex_from_1_simplices_truncated (SimplicialComplex* complex,
140 int n, int k) {
141     int i,j;
142     int** M = (int**) malloc(n*sizeof(int*));
143     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
138 // nomenclatura ed evitare di scorrere sempre la lista)
139 int* vertices = (int*) malloc(n*sizeof(int));
140 Simplex* current=complex[0].simplices;
141 i=0;
142 while (current!=NULL) {
143     vertices[i]=current->vertices[0];
144     i++;
145     current=current->next;
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-
152         // semplice
153         if (check_neighbor(complex,vertices[i],vertices[j])!=0) {
154             M[i][j]=1; M[j][i]=1;
155         }
156         else {
157             M[i][j]=0; M[j][i]=0;
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-
165 // simplessi specificati dalla matrice di adiacenza M n*n.
166 // E' necessario che il complesso sia ordinato e k>0. M[i][j] sara' diversa da 0 se
167 // (i,j) e' nel complesso (denomino i vertici a partire da 0).
168 void add_k_simplices_adjacency_matrix (SimplicialComplex* complex, int** M, int n,
169 int k) {
170     // parto con 0 k-simplessi: la lista complex[k].simplices e' vuota
171     complex[k].simplices=NULL;
172     int num_k_simplices=0;
173     int i,j;
174
175     // ricerco i k-simplessi da aggiungere:
176
177     Simplex* previous_simplex = complex[k-1].simplices; // scorrerà i (k-1)-
178     // simplessi
179     Simplex *current = NULL, *new_simplex; // current sarà un puntatore all'ultimo
180     // elemento della lista dei k-simplessi, new_simplex servirà per creare quelli
181     // da aggiungere
182     while (previous_simplex!=NULL) { // per costruire i k-simplessi devo partire
183     // dai (k-1)-simplessi
184
185         // cerco nuovi vertici a partire da quelli successivi all'ultimo gia'
186         // presente (funziona se il complesso e' ordinato)
187         for (j=previous_simplex->vertices[k-1]+1; j<n; j++) {
188             if (check_neighbor_adjacency_matrix(M,previous_simplex->vertices,k,j)
189             !=0) {
190                 // ho trovato il vicino j: costruisco il nuovo k-simplesso
191
192                 if (num_k_simplices==0) { // caso in cui aggiungo il primo k-
193                 // semplice alla lista
194                     new_simplex = (Simplex*) malloc(sizeof(Simplex));
195                     new_simplex->next = NULL;
196                     new_simplex->position = num_k_simplices;
197                     new_simplex->vertices = (int*) malloc((k+1)*sizeof(int));
198                     for (i=0; i<k; i++) new_simplex->vertices[i]=previous_simplex->
199                     vertices[i];
200                     new_simplex->vertices[k]=j;
201
202                     complex[k].simplices=new_simplex;
203                     current=new_simplex;
204                 }
205             }
206         }
207     }
208 }

```

```

194         else { // caso in cui la lista dei k-simplessi e' non vuota
195             new_simplex = (Simplex*) malloc(sizeof(Simplex));
196             new_simplex->next = NULL;
197             new_simplex->position = num_k_simplices;
198             new_simplex->vertices = (int*) malloc((k+1)*sizeof(int));
199             for (i=0; i<k; i++) new_simplex->vertices[i]=previous_simplex->
                vertices[i];
200             new_simplex->vertices[k]=j;
201
202             current->next=new_simplex;
203             current=current->next;
204         }
205         num_k_simplices++;
206     }
207 }
208
209 previous_simplex = previous_simplex->next;
210 }
211
212 complex[k].size=num_k_simplices;
213 return;
214 }
215
216 // Controlla se il vertice di indice j e' "vicino" ai k vertici di indici rows,
217 // ossia se il minore di righe rows e colonna j e' tutto non nullo.
218 // Restituisce 0 se j non e' un vicino, 1 se lo e'.
219 int check_neighbor_adjacency_matrix (int** M, int* rows, int k, int j) {
220     for (int i=0; i<k; i++) {
221         if (M[rows[i]][j]==0) return 0;
222     }
223     return 1;
224 }
225
226 // Controlla se a e b sono "vicini", ossia se la coppia (a,b) appartiene agli 1-
227 // semplici. Restituisce 0 se non sono vicini, 1 se lo sono.
228 int check_neighbor (SimplicialComplex* complex, int a, int b) {
229     Simplex* current=complex[1].simplices;
230     while (current!=NULL) { // scorro lungo gli 1-simplessi
231         if ((current->vertices[0]==a && current->vertices[1]==b) || (current->
            vertices[0]==b && current->vertices[1]==a)) return 1;
232         current=current->next;
233     }
234     return 0;
235 }

```

1.2 Complessi di catene e omologia simpliciale

Per studiare meglio i complessi simpliciali 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

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

dove con \hat{x}_i si intende che l'elemento i -esimo del simpleso 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 gruppoid 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

```

1 #pragma once
2 // MioFile.h
3 #ifndef Complesso_Simplciale
4     #define Complesso_simpliciale
5
6     #include <stdbool.h>
7     #include <stdio.h>
8     #include <stdlib.h>
9     #include <math.h>    // Per la funzione pow
10
11     /*Il complesso simpliciale è formato da un vettore di liste di vettori
12     Ogni cella del primo vettore conterrà i rispettivi semplici cella 0 gli 0-
13     simplessi cella 1 gli 1-simplessi etc..
14     a loro volta gli n-simplessi sono organizzati in liste di vettori così da
15     distinguere i vertici che compongono il rispettivo simpleso.*/
16
17     //Definizione delle strutture
18     //Definizione degli n-simplessi
19     typedef struct Simplex {
20         int* vertices; // Array di vertici del simplicio
21         int position; // Ordine del simplicio nella base
22         struct Simplex* next;
23     } Simplex;
24
25     // Definizione di un complesso simpliciale
26     typedef struct {
27         Simplex* simplices; // Array di simplici
28         int size; // Numero degli n-simplessi essenziale per costruire le
29                 // matrici di bordo e non scorrere tutta la lista
30     } SimplicialComplex;
31
32     //Definizione delle funzioni
33     Simplex* createSimplex(int, int);
34     Simplex* createSimplex_file(int, int, FILE*);
35     SimplicialComplex* readComplex(int);
36     SimplicialComplex* readComplex_file(char *);
37     void printComplex(SimplicialComplex*, int);
38     int* creasubs(int*, int, int);
39     bool equal(int*, int*, int);
40     bool isIn(Simplex*, Simplex*, int);
41     bool isSimplicial(SimplicialComplex*, int);
42     int** edge_Matrix(SimplicialComplex*, int);
43     int base_number(Simplex*, int*, int);
44     int matrix_rank(int**, int, int);
45
46     //Esplicitare le funzioni
47
48     #pragma region gestione dei semplici e dei complessi simpliciali
49
50     // funzione per leggere un complesso da input quindi sappiamo la misura del
51     // complesso più grande = size
52     SimplicialComplex* readComplex(int size){
53         SimplicialComplex* complex = (SimplicialComplex*)malloc(size *
54             sizeof(SimplicialComplex));
55         int n = 0;
56
57         //itero i vari n-simplessi su tutta la lunghezza del vettore
58         for (int i = 0; i < size; i++) {
59             printf("Inserisci il numero di %d-simplessi: ", i);
60             scanf("%d", &n);
61             complex[i].size = n;
62             complex[i].simplices = createSimplex(i, n);
63         }
64         return complex;
65     }
66
67     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(
76             size * sizeof(SimplicialComplex));
77         int n = 0;
78         for (int i = 0; i < size; i++) {
79             fscanf(f, "%d", &n);
80             complex[i].size = n;
81             complex[i].simplices = createSimplex_file(i, n, f);
82         }
83         fclose(f);
84         return complex;
85     }
86
87 // funzione per stampare un semplice
88 void printComplex(SimplicialComplex* complex, int size){
89     Simplex* app;
90     for (int i = 0; i < size; i++) {
91         printf("%d-simplessi\n", i);
92         printf("size = %d\n", complex[i].size);
93         app = complex[i].simplices;
94         if (app) {
95             printf("{ ");
96             while (app) {
97                 printf("(" ");
98                 for (int j = 0; j <= i; j++)
99                     printf("%d ", app->vertices[j]);
100                 printf(")");
101                 app = app->next;
102                 if (app)
103                     printf(", ");
104             }
105             printf(" }\n");
106         }
107     }
108     return;
109 }
110
111 // funzione per creare un sottoinsieme di un semplice togliendo il vertice
112 // i-esimo
113 int* creasubs(int* v, int i, int n) {
114     int* s = (int*)malloc((n) * sizeof(int));
115     int k = 0;
116     //porto il ciclo fino a n perchè il vertice i-esimo non deve essere
117     //inserito quindi inserisco solo n-1 vertici
118     for (int j = 0; j <= n; j++) {
119         if (j != i) {
120             s[k] = v[j];
121             k++;
122         }
123     }
124     return s;
125 }
126
127 //controlla se due vettori sono uguali
128 bool equal(int* v, int* s, int n) {
129     for (int i = 0; i < n; i++) {
130         if (v[i] != s[i])
131             return false;
132     }
133     return true;
134 }

```

```

132
133 //Funzione per controllare se i sottoinsiemi dell' n simpleso sono
    contenuti nell' n-1 simpleso
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-simpleso
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-simpleso 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 simpleso 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 simpleso 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 insiemi di ciascun semplice e
245             //vedo che posizione ha nella base
246             for (int j = 0; j <= n; j++) {
247                 v = creasubs(Nsimp->vertices, j, n);
248                 k = base_number(Nmosimp, v, n);
249                 //printf("k = %d\n", k);
250                 if (k != -1) {
251                     matrix[k][i] = pow(-1, j);
252                 }
253             }
254             Nsimp = Nsimp->next;
255         }
256         return matrix;
257     }
258
259     int base_number(Simplex* sc, int* v, int n) {
260         Simplex* app = sc;
261         while (app) {
262             if (equal(app->vertices, v, n))
263                 return app->position;
264             app = app->next;
265         }
266         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
298             // scambiare
299             int reduce = 1;
300             for (int i = row + 1; i < rows; i++) {
301                 if (fabs(matrix[i][row]) > 0) {
302                     swap_rows(matrix, row, i, cols);
303                     reduce = 0;
304                     break;
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 $Im(\partial : C_{k+1} \rightarrow C_k) \subset Ker(\partial : C_k \rightarrow C_{k-1})$, quindi ha senso introdurre il concetto di **omologia del complesso simpliciale**:

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

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

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

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

Listing 5: Omologie.c

```

1 #include "..\Include\Compl_Simpl.h"
2 #include "..\Include\Omo.h"
3
4 // Funzione per calcolare l'omologia di un complesso simpliciale
5 int main() {
6
7     int size = 0, n = 2;
8     int** matrix1 = NULL, ** matrix2 = NULL;
9     //mi serve per stampare il complesso simpliciale
10    printf("Inserisci la grandezza del semplice piu' grande nel complesso: ");
11    scanf("%d", &size);
12
13    //Leggo il semplice da un file e lo stmpo
14    SimplicialComplex* complex = readComplex_file("simplicial_complex.txt");
15    printComplex(complex, 3);
16
17    //controllo sia un complesso simpliciale
18    if (!isSimplicial(complex, size)) {
19        printf("Errore il complesso non e' simpliciale inserire un nuovo
20            semplice\n");
21        return 1;
22    }
23
24    //chiedo di quale numero voglio calcolare h
25    printf("Inserisci il numero n di cui vuoi calcolare l'omologia H_n \n (n >=
26        0) = ");
27    scanf("%d", &n);
28
29    n++;
30
31    //calcolo le matrici di bordo dei semplici n e n-1
32    //per comodità di notazione ho aggiunto 1 ad n
33    if (n == 1) {
34        matrix2 = NULL;
35        matrix1 = edge_Matrix(complex, n);
36    }
37    else if (n == size) {
38        matrix1 = NULL;
39        matrix2 = edge_Matrix(complex, n - 1);
40    }
41    else if (n >= size + 1) {
42        matrix1 = NULL;
43        matrix2 = NULL;
44    }
45    else {
46        matrix1 = edge_Matrix(complex, n);
47        matrix2 = edge_Matrix(complex, n - 1);
48    }
49
50    //calcolo la dimensione dell'omologia più veloce che decomporla in z-moduli
51    int h = dim_omologia(matrix1, complex[n - 1].size, complex[n].size, matrix2,
52        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 dell'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     //semplice 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
42     //input solo le due matrici di bordo
43     int dim_omologia(int** matrix1, int row1, int col1, int** matrix2, int row2
44     , int col2) {
45         int h = 0, ck = 0, rk = 0, rk1 = 0;
46
47         //controlliamo che le due matrici diano un complesso
48         if (matrix1 && matrix2) {
49             if (!is_Complex(matrix1, row1, col1, matrix2, row2, col2))
50             {
51                 return -1;
52             }
53         }
54
55         int** D = NULL;
56         int** S = NULL;
57         int** T = NULL;
58
59         //calcoliamo la dimensione dell'omologia come  $h = ck - rk - rk1$ 
60         //dove ck è il numero di n semplici
61         //rk è il rango della matrice di bordo n+1-esima e rk1 è il rango
62         //della matrice di bordo n-esima
63         //controlliamo se esistono le matrici e quindi ne calcolo il rango
64         //utilizzando la matrice di smith
65         if (matrix2) {
66
67             D = input_null(D, row2, col2);
68             S = input_id(S, row2);
69             T = input_id(T, col2);
70
71             for (int i = 0; i < row2; i++) {
72                 for (int j = 0; j < col2; j++) {
73                     D[i][j] = matrix2[i][j];
74                 }
75             }
76
77             ck = col2;
78
79             SmithNormalForm(D, S, T, row2, col2);
80             rk = rank_matrix_diag(D, row2, col2);
81             printf("rk = %d\n", rk);
82
83             free(D);
84             free(S);
85             free(T);
86         }
87         else
88             rk = 0;
89
90         if (matrix1) {
91             D = input_null(D, row1, col1);
92             S = input_id(S, row1);
93             T = input_id(T, col1);
94
95             for (int i = 0; i < row1; i++) {
96                 for (int j = 0; j < col1; j++) {
97                     D[i][j] = matrix1[i][j];
98                 }
99             }
100
101             ck = row1;
102
103             SmithNormalForm(D, S, T, row1, col1);
104             rk1 = rank_matrix_diag(D, row1, col1);
105             printf("rk1 = %d\n", rk1);

```

```

101         free(D);
102         free(S);
103         free(T);
104     }
105     else
106         rk1 = 0;
107
108
109     printf("ck = %d\n", ck);
110     h = ck - rk - rk1;
111
112     return h;
113 }
114
115
116
117 //Come rappresentare l'omologia in z-moduli
118 int** omologia(int** matrix1, int row1, int col1, int** matrix2, int row2,
119               int col2) {
120     int** h = NULL;
121
122     //controlliamo che le due matrici diano un complesso
123     if (!is_Complex(matrix1, row1, col1, matrix2, row2, col2)) {
124         printf("Errore: le matrici non formano un complesso\n");
125         return NULL;
126     }
127
128     int** D = NULL;
129     D = input_null(D, row2, col2);
130     int** S = NULL;
131     S = input_id(S, row2);
132     int** T = NULL;
133     T = input_id(T, col2);
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
145     //ovvero
146     //le ultime r colonne della matrice T (SAT = D) che salvo in T2
147     int r = rank_matrix_diag(D, row2, col2);
148
149     int** T2 = NULL;
150     T2 = input_null(T2, col2, col2-r);
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     //troviamo la forma di smith della base del nucleo appena trovata
159     //per risolvere Bx = An+1 negli interi moltiplicando per l'inverso
160     //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 attraverso
200         smith e la matrice diagonale
201
202     S1 = NULL;
203     S1 = input_id(S1, col2 - r);
204     T1 = NULL;
205     T1 = input_id(T1, col2 - r);
206     D1 = NULL;
207     D1 = input_null(D1, col2 - r, col2 - r);
208
209     for (int i = 0; i < col2 - r; i++) {
210         for (int j = 0; j < col2 - r; j++) {
211             D1[i][j] = temp[i][j];
212         }
213     }
214
215     SmithNormalForm(D1, S1, T1, col2 - r, col2 - r);
216     h = (int*)calloc(col2 - r - 1, sizeof(int));
217     for (int i = 0; i < col2 - r; i++) {
218         h[i] = D1[i][i];
219     }
220
221     //printiamo il risultato ottenuto ovvero i valori di h
222     printf("\nDecomposizione di h\n");
223     printf("Z_%d", h[0]);
224     for (int i = 1; i < col2 - r; i++) {
225         printf(" + ");
226         printf("Z_%d", h[i]);
227     }
228
229     /* non sempre trovo un minore invertibile negli itneri quindi
230         questo codice non va bene
231
232     int** B = NULL;
233     int** B_inv = NULL;
234     int** C = NULL;
235     B = input_null(B, col2 - r, col2 - r);
236     B_inv = input_null(B_inv, col2 - r, col2 - r);
237     C = input_null(C, col2 - r, col1);

```

```

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

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

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
122         // per invertire le colonne bisogna lavorare sulla trasposta
123         Tt=traspMatrix(T, Tt, n, n); // invertire le colonne è un'
124         // operazione che lavora su T
125         for (int j=k; j<(n-1); j++) {
126             swapRows(A, j, j+1, m); // inverte le righe j e j+1 di una
127             // matrice
128             swapRows(Tt, j, j+1, n);
129         }
130         A=traspMatrix(At, A, n, m); // trasposta di trasposta per tornare
131         // nella forma originaria
132         T=traspMatrix(Tt, T, n, n);
133     }
134     return;
135 }
136
137 // invertire i segni di una riga della matrice
138 void switchSigns (int **A, int **mat, int c, int m, int n, int k) {
139     for (int i=k; i<m; i++) {
140         if (A[i][k]<0) {
141             for (int j=k; j<n; j++) {
142                 A[i][j]=-A[i][j]; // matrice A
143             }
144             for (int l=0; l<c; l++) {
145                 mat[i][l]=-mat[i][l]; // matrice S o T a seconda
146                 // dei casi
147             }
148         }
149     }
150     return;
151 }

```

```

147
148 // scambiare 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
177     // portare il minimo in alto
178     swapRows(mat, k, r_min, c); // stesso lavoro anche su S o T a seconda dei
179     // casi
180     return;
181 }
182
183 // divisione con i resti
184 void div_rem (int **A, int **mat, int c, int m, int n, int k) {
185     int quot=0;
186     for (int i=k+1; i<m; i++) {
187         quot=A[i][k]/A[k][k]; // individuare il quoziente della divisione
188         addRows(A, k, i, quot, n); // modificare le righe di A (inserendo i
189         // resti)
190         addRows(mat, k, i, quot, c); // stesso lavoro su S o T a seconda
191         // dei casi
192     }
193     return;
194 }
195
196 // verificare se la colonna è della forma [0, ..., 0, d, 0, ..., 0] con d in
197 // posizione k
198 bool nullVector (int **mat, int m, int k) {
199     for (int i=0; i<m; i++) {
200         if (i!=k && mat[i][k]!=0) {
201             return false;
202         }
203     }
204     return true;
205 }
206
207 // prima forma di Smith che porta a un matrice diagonale ma i cui elementi in
208 // diagonale non necessariamente sono multipli
209 void Smith_fst (int **A, int **S, int **T, int **At, int **Tt, int m, int n, int k)
210 {
211     while(!nullVector(A, m, k) || !nullVector(traspMatrix(A, At, m, n), n, k))
212     { //lavoro sulla riga e colonna k fino a che entrambe non siano
213       //contemporaneamente della forma [d, 0, ..., 0]
214       switchZeros(A, T, At, Tt, m, n, k); // per prima cosa sposto l'
215       //eventuale colonna nulla
216       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
           div_rem(A, S, m, m, n, k); // divisione con i resti
210         }
211     }
212     // per poter lavorare sulle righe traspongo la matrice e ripeto lo
           stesso procedimento fatto per la colonna
213     At=traspMatrix(A, At, m, n);
214     Tt=traspMatrix(T, Tt, n, n);
215     while (!nullVector(A, n, k)) {
216         switchSigns(A, Tt, n, n, m, k);
217         while (!nullVector(A, n, k)) {
218             minUp(A, Tt, n, n, m, k);
219             div_rem(A, Tt, n, n, m, k);
220         }
221     }
222     A=traspMatrix(A, A, n, m);
223     T=traspMatrix(Tt, T, n, n);
224 }
225 switchSigns(A, S, m, m, n, k); // cambio il segno dell'ultimo elemento in
           diagonale nel caso sia negativo
226 return;
227 }
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(A, n, m);
270     int **Tt=NULL;
271     Tt=input_null(T, 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
319 // colonna j di B
320 for (int k=0; k<n; k++) {
321     C[i][j]=C[i][j]+A[i][k]*B[k][j];
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
342 // funzione SmithNormalForm, l'unica differenza è che produce anche le matrici
343 // S_inv e T_inv tali per cui S_inv*T_inv=A (di seguito andiamo quindi a
344 // commentare solo le differenze rispetto alla funzione precedente)
345 void SmithNormalForm5mat (int **A, int **S, int **T, int **S_inv, int **T_inv, int
346 m, int n) {
347     int **At=NULL;
348     At=input_null(AT, n, m);
349     int **Tt=NULL;
350     Tt=input_null(Tt, n, n);
351     int **SinvT=NULL;
352     SinvT=input_null(SinvT, m, m);
353
354     int r=my_min(m, n);
355
356     for (int k=0; k<r; k++) {
357         Smithfst5mat(A, S, T, S_inv, T_inv, At, Tt, SinvT, m, n, k);
358     }
359
360     int err=checkDiagZeros(A, r);
361     if (err!=r) {
362         SinvT=traspMatrix(S_inv, SinvT, m, m);
363         for (int j=err+1; j<r; j++) {
364             addRows(A, j, err, -1, n);
365             addRows(S, j, err, -1, m);
366             addRows(SinvT, err, j, 1, m); // presa S1 e S2, mentre S=S2
367             // *S1, S_inv=S1_inv*S2_inv quindi così come per le T per
368             // moltiplicare a destra usavo la trasposta lo stesso
369             // faccio con S_inv (tenendo conto che l'inversa fa i
370             // passaggi al contrario, quindi anzichè -1 metto 1 e
371             // invertito it e i)
372         }
373         S_inv=traspMatrix(SinvT, S_inv, m, m);
374         for (int k=err; k<r; k++) {
375             Smithfst5mat(A, S, T, S_inv, T_inv, At, Tt, SinvT, m, n, k
376 );
377         }
378     }
379
380     for (int it=0; it<r; it++) {
381         if (A[it][it]==0) {
382             break;
383         }
384         if (!bool_dk(A, r, it)) {
385             SinvT=traspMatrix(S_inv, SinvT, m, m);
386             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,
384                     m, n, iter);
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
394 // di S_inv e T_inv
395 void Smith_fst5mat (int **A, int **S, int **T, int **S_inv, int **T_inv, int **At,
396 int **Tt, int **SinvT, int m, int n, int k) {
397     while(!nullVector(A, m, k) || !nullVector(traspMatrix(A, At, m, n), n, k))
398     {
399         switchZeros5mat(A, T, T_inv, At, Tt, m, n, k);
400         SinvT=traspMatrix(S_inv, SinvT, m, m);
401         while (!nullVector(A, m, k)) {
402             switchSigns5mat(A, S, SinvT, m, m, n, k);
403             while (!nullVector(A, m, k)) {
404                 minUp5mat(A, S, SinvT, m, m, n, k);
405                 div_rem5mat(A, S, SinvT, m, m, n, k);
406             }
407         }
408         S_inv=traspMatrix(SinvT, S_inv, m, m);
409         At=traspMatrix(A, At, m, n);
410         Tt=traspMatrix(T, Tt, n, n);
411         while (!nullVector(A, m, k)) {
412             switchSigns5mat(A, S, SinvT, m, m, n, k);
413             while (!nullVector(A, m, k)) {
414                 minUp5mat(A, S, SinvT, m, m, n, k);
415                 div_rem5mat(A, S, SinvT, m, m, n, k);
416             }
417         }
418         A=traspMatrix(A, At, m, n);
419         T=traspMatrix(Tt, T, n, n);
420     }
421     SinvT=traspMatrix(S_inv, SinvT, m, m);
422     switchSigns5mat(A, S, SinvT, m, m, n, k);
423     S_inv=traspMatrix(SinvT, S_inv, m, m);
424     return;
425 }
426
427 // funzione analoga a switchZeros con l'aggiunta di S_inv e T_inv
428 void switchZeros5mat (int **A, int **T, int **T_inv, int **At, int **Tt, int m, int
429 n, int k) {
430     bool var=true;
431     for (int i=k; i<m; i++) {
432         if (A[i][k]!=0) {
433             var=false;
434             break;
435         }
436     }
437     if (var) {
438         At=traspMatrix(A, At, m, n);
439         Tt=traspMatrix(T, Tt, n, n);
440         for (int j=k; j<(n-1); j++) {
441             swapRows(A, j, j+1, m);
442             swapRows(Tt, j, j+1, n);
443             swapRows(T_inv, j, j+1, n);
444         }
445         A=traspMatrix(A, At, m, n);

```



```

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
457                                     seconda dei casi
458             }
459         }
460     }
461     return;
462 }
463
464 // funzione analoga a minUp con l'aggiunta di S_inv e T_inv
465 void minUp5mat (int **A, int **mat, int **mat2, int c, int m, int n, int k) {
466     int min=A[k][k], r_min=k;
467     for(int i=k; i<m; i++) {
468         if (A[i][k]<min && A[i][k]!=0 || min==0) {
469             min=A[i][k];
470             r_min=i;
471         }
472     }
473     swapRows(A, k, r_min, n);
474     swapRows(mat, k, r_min, c);
475     swapRows(mat2, k, r_min, c); // stesso lavoro anche su S_inv o T_inv a
476     seconda dei casi
477     return;
478 }
479
480 // funzione analoga a div_rem con l'aggiunta di S_inv e T_inv
481 void div_rem5mat (int **A, int **mat, int **mat2, int c, int m, int n, int k) {
482     int quot=0;
483     for (int i=k+1; i<m; i++) {
484         quot=A[i][k]/A[k][k];
485         addRows(A, k, i, quot, n);
486         addRows(mat, k, i, quot, c);
487         addRows(mat2, i, k, -quot, c); // stesso lavoro su S_inv o T_inv a
488         seconda dei casi (l'inversa fa i passaggi al contrario)
489     }
490     return;
491 }
492 #endif

```

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

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

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

1.4 Categorie, funtori e moduli di persistenza

Nello studio dei dati che ci siamo posti ad inizio corso molto spesso bisogna tenere conto che le informazioni sono variabili nel tempo (aumentano i dati, si modificano nel tempo etc.). Per studiare

questa variante più formalmente si introducono i concetti di **categoria**, ovvero un insieme di oggetti (nel nostro caso specifici complessi simpliciali, complessi di moduli o spazi vettoriali), e di **funtore**, una applicazione $F : C \rightarrow D$ tra due categorie con le seguenti proprietà:

1. $\forall f : X \rightarrow Y \text{ in } C \ F(f) : F(X) \rightarrow F(Y) \text{ in } D$
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 $\{t_0, t_1, \dots, t_n\}$ 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:

$$(\{t_0, \dots, t_n\}, \leq) \rightarrow \text{insiemi simpliciali} \xrightarrow{C(\cdot, \mathbb{Q})} \text{complessi di moduli} \xrightarrow{H_\cdot} \text{spazi vettoriali},$$

ovvero come variano nel tempo i dati forniti.

Poiché 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 $\{t_0 \leq t_1 \leq \dots \leq t_n\}$. 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

$$\begin{aligned} \beta_{i,j} &= rg(\varphi_{i \rightarrow j}), \\ \mu_{i,j} &= \beta_{i,j+1} + \beta_{i-1,j} - \beta_{i,j} - \beta_{i-1,j-1}. \end{aligned}$$

Listing 8: Barcode.h

```

1  #pragma once
2  #ifndef Barcode
3      #define Barcode
4      #include "Complesso_da_1_simplessi.h"
5      #include "Smith.h"
6      #include "Matrix.h"
7
8      /*Come creiamo il barcode
9      come prima cosa creiamo il modulo di persistenza ovvero la collezione dell'
10     evoluzione dei nostri complessi simpliciali
11     a questo punto ci chiediamo di quale H vogliamo calcolare il barcode e
12     calcoliamo le rispettive matrici di bordo
13     poi calcoliamo una base del nucleo
14     applichiamo la trasformazione da H a H+t
15     calcoliamo il rango di questa applicazione fusa con la matrice di bordo
16     e sottraiamo il rango della matrice di bordo il risultato è l'elemento per
17     il resto dobbiamo avere che j >= i e calcoliamo tutti gli altri
18     elementi*/
19
20     //Definizione modulo di persistenza
21
22     /*Definisco l'evoluzione delle matrici in base al cambiamento delle
23     distanze
24     ovvero mi salvo una lista di matrici di zero e 1 che sono le matrici di
25     adiacenza */
26     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'
31     // evoluzione
32     typedef struct Modulo_Persistenza {
33         double l_min;
34         double l_max;
35         int size;
36         SimplicialComplex* sc;
37         struct Modulo_Persistenza* next;
38         struct Modulo_Persistenza* prev;
39     } M_P;
40
41     //matrici_persistenza* create_M_P(void);
42     M_P* create_Modulo_Persistenza(matrici_persistenza*, int);
43     int** matrix_phi_l1_to_l2(Simplex*, int, Simplex*, int, int);
44     int beta_i_j(M_P*, double, double, int, int);
45     int** beta_matrix(int, int, double, M_P*, int, int);
46     int** mu_matrix(int**, int);
47     double** distance_matrix(int**, int);
48     int** input_point(int*);
49     int** zero_one_matrix(double**, int, double);
50     double* find_lambda_value(double**, int);
51     matrici_persistenza* create_matrix_persistenza(double**, int, double*);
52     int compare_double(const void*, const void*);
53     void printl(M_P*, int);
54
55     /* Esempio non da input
56     matrici_persistenza* create_M_P(void) {
57         matrici_persistenza* mp1 = (matrici_persistenza*)malloc(sizeof(
58             matrici_persistenza));
59         matrici_persistenza* mp2 = (matrici_persistenza*)malloc(sizeof(
60             matrici_persistenza));
61         matrici_persistenza* mp3 = (matrici_persistenza*)malloc(sizeof(
62             matrici_persistenza));
63         matrici_persistenza* mp4 = (matrici_persistenza*)malloc(sizeof(
64             matrici_persistenza));
65         mp1->l_min = 0.5;
66         mp1->l_max = 1;
67         mp2->l_min = 1.5;
68         mp2->l_max = 10;
69         mp3->l_min = 10.5;
70         mp3->l_max = 10.5;
71         mp4->l_min = 11;
72         mp4->l_max = 11; // per segnalare la fine
73         mp1->size = 3;
74         mp2->size = 3;
75         mp3->size = 3;
76         mp4->size = 3;
77         int** m1 = NULL, ** m2 = NULL, ** m3 = NULL, ** m4 = NULL;
78         m1 = input_null(m1, 3, 3);
79         m2 = input_null(m2, 3, 3);
80         m3 = input_null(m3, 3, 3);
81         m4 = input_null(m4, 3, 3);
82         m1[0][0] = 1; m1[0][1] = 0; m1[0][2] = 0;
83         m1[1][0] = 0; m1[1][1] = 1; m1[1][2] = 0;
84         m1[2][0] = 0; m1[2][1] = 0; m1[2][2] = 1;
85         m2[0][0] = 1; m2[0][1] = 1; m2[0][2] = 0;
86         m2[1][0] = 1; m2[1][1] = 1; m2[1][2] = 0;
87         m2[2][0] = 0; m2[2][1] = 0; m2[2][2] = 1;
88         m3[0][0] = 1; m3[0][1] = 1; m3[0][2] = 0;
89         m3[1][0] = 1; m3[1][1] = 1; m3[1][2] = 1;
90         m3[2][0] = 0; m3[2][1] = 1; m3[2][2] = 1;
91         m4[0][0] = 1; m4[0][1] = 1; m4[0][2] = 1;

```

```

87         m4[1][0] = 1; m4[1][1] = 1; m4[1][2] = 1;
88         m4[2][0] = 1; m4[2][1] = 1; m4[2][2] = 1;
89         mp1->matrix_d = m1;
90         mp2->matrix_d = m2;
91         mp3->matrix_d = m3;
92         mp4->matrix_d = m4;
93         mp1->next = mp2;
94         mp2->next = mp3;
95         mp3->next = mp4;
96         mp4->next = NULL;
97         mp1->prev = NULL;
98         mp2->prev = mp1;
99         mp3->prev = mp2;
100        mp4->prev = mp3;
101        return mp1;
102    }*/
103
104    /*Data la successione di matrici creo da quelle i complessi simpliciali
105    salvandomi ogni volta il lambda min e max in cui quel complesso non varia
106    utilizzo la fomra troncata per risparmiare memoria e per alcuni
    accorgimenti detti all'utente*/
107    M_P* create_Modulo_Persistenza(matrici_persistenza* mp, int max) {
108        M_P* mod_p = NULL, * app = NULL, * my_new = NULL;
109        int size = mp->size;
110        mod_p = (M_P*)malloc(sizeof(M_P));
111        mod_p->l_min = mp->l_min;
112        mod_p->l_max = mp->l_max;
113        mod_p->size = size;
114        mod_p->sc = complex_from_adjacency_matrix_truncated(mp->matrix_d,
115        size, max);
116        mod_p->prev = NULL;
117        app = mod_p;
118
119        while (mp->next) {
120            mp = mp->next;
121            size = mp->size;
122            my_new = (M_P*)malloc(sizeof(M_P));
123            my_new->l_min = mp->l_min;
124            my_new->l_max = mp->l_max;
125            my_new->sc = complex_from_adjacency_matrix_truncated(mp->
126            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
138    semplici
139    int** matrix_phi_l1_to_l2(Simplex* s1, int sizeb1, Simplex* s2, int sizeb2,
140    int size) {
141        int** matrix = NULL;
142        matrix = input_null(matrix, sizeb2, sizeb1);
143        Simplex* current = s1;
144        int col = 0;
145        //scorro il primo semplice che è contenuto nel secondo per
146        costruzione e mi salvo il numero
147        //corrispondente alla posizione e creo la matrice di cambiamento di
148        base
149        while (current)
150        {
151            int* v = current->vertices;
152            int k = base_number(s2, v, size);
153            if (k != -1) {
154                matrix[k][col] = 1;
155            }
156        }
157    }

```

```

150         current = current->next;
151         col++;
152     }
153
154     return matrix;
155 }
156
157 //calcolo l'elemento ij della matrice beta sempre i > j
158 //n è il grado del complesso i e j l'evoluzione al tempo i e j
159 //h l'omologia cercata
160 int beta_i_j(M_P* mp, double i, double j, int h, int n) {
161     SimplicialComplex* sc1 = NULL, * sc2 = NULL;
162     M_P* app = mp;
163
164     if (i < app->l_min)
165         return 0;
166     if(h > n)
167         return 0;
168     //come prima cosa trovo a quale complessi simpliciale appartengono
169     //le evoluzioni i e j
170     while (app && !sc2) {
171         if (app->l_min <= i && i <= app->l_max) {
172             sc1 = app->sc;
173         }
174         if (app->l_min <= j && j <= app->l_max) {
175             sc2 = app->sc;
176         }
177         app = app->next;
178     }
179     //se trovo il secondo anche il primo esiste per costruzione
180     //altrimenti torno 0
181     if (!sc2)
182         return 0;
183
184     //se il primo ha dimensione nulla torno zero perchè vuole dire che
185     //non ci sono evoluzioni
186     if (sc1[h].size == 0)
187         return 0;
188
189     //printf("sc1[h].size = %d\n", sc1[h].size);
190
191     //se tutto è andato bene calcolo il rango dell'omologia tramite phi
192     //per prima cosa trovo la matrice di cambio di base
193     int** phi_i_j = matrix_phi_l1_to_l2(sc1[h].simplices, sc1[h].size,
194                                         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 semplici h e h -1
202
203     //quindi se h = 0 la matrice è nulla altrimenti la calcolo e
204     //calcolo una base del nucleo
205     if (h == 0) {
206         //edge1 = input_id(edge1, sc1[h].size);
207         edge1 = input_null(edge1, sc1[h].size, sc1[h].size);
208         //print_matrix(edge1, sc1[h].size, sc1[h].size);
209         kernel_edge1 = input_id(kernel_edge1, sc1[h].size);
210         colonne_base_rango = sc1[h].size;
211     }
212     else {
213         edge1 = edge_Matrix(sc1, h);
214         kernel_edge1 = kernel_base(edge1, sc1[h - 1].size, sc1[h].
215                                     size, &colonne_base_rango);
216     }
217 }

```

```

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

```

```

277 // k è il grado del complesso e h è l'omologia richiesta
278 int** beta_matrix(int min, int max, double passo, M_P* mp, int h, int k) {
279     int** matrix = NULL;
280     //numero di passi da calcolare in base al lambda massimo e minimo
281     int n = ((max - min) / passo) + 1;
282     matrix = input_null(matrix, n + 1, n + 1);
283     int i = 0;
284     int j = 0;
285
286     for (i = 0; i <= n; i++) {
287         for (j = i; j <= n; j++) {
288             matrix[i][j] = beta_i_j(mp, min + i * passo, min +
                                     j * passo, h, k);
289         }
290     }
291     return matrix;
292 }
293
294 //mu è una combinazione lineare delle entrate di beta
295 int** mu_matrix(int** beta, int n) {
296     int i = 0, j = 0;
297     int** matrix = NULL;
298     matrix = input_null(matrix, n, n);
299     for (i = 1; i < n; i++)
300         for (j = i + 1; j < n; j++)
301             matrix[i][j] = beta[i][j - 1] - beta[i][j] + beta[i
                                     - 1][j] - beta[i - 1][j - 1];
302     return matrix;
303 }
304
305 //calcolo la matrice delle distanze tra i punti nel piano cartesiano
306 double** distance_matrix(int** point, int n) {
307     int i = 0, j = 0;
308     double** matrix = NULL;
309     matrix = (double**)calloc(n, sizeof(double*));
310     for (i = 0; i < n; i++) {
311         matrix[i] = (double*)calloc(n, sizeof(double));
312     }
313     for (i = 0; i < n; i++) {
314         for (j = i; j < n; j++) {
315             matrix[i][j] = sqrt(pow(point[i][0] - point[j][0],
                                     2) + pow(point[i][1] - point[j][1], 2));
316             matrix[j][i] = matrix[i][j];
317         }
318     }
319     return matrix;
320 }
321
322 //richiede i punti da input
323 int** input_point(int *k) {
324     int** matrix = NULL;
325     int n = 0;
326     printf("Inserisci il numero di punti: n = ");
327     scanf("%d", &n);
328     *k = n;
329     matrix = input_null(matrix, n, 2);
330     int i = 0;
331     for (i = 0; i < n; i++) {
332         printf("Inserisci le coordinate del punto %d: ", i + 1);
333         scanf("%d %d", &matrix[i][0], &matrix[i][1]);
334     }
335
336     return matrix;
337 }
338
339 //calcola le matrici 0-1 partendo dalla matrice delle distanze e un valore
    lambda
340 int** zero_one_matrix(double** matrix, int n, double lambda) {
341     int** m = NULL;
342     m = input_null(m, n, n);

```

```

343     int i = 0, j = 0;
344     for (i = 0; i < n; i++) {
345         for (j = i; j < n; j++) {
346             if (matrix[i][j] < lambda) {
347                 m[i][j] = 1;
348                 m[j][i] = 1;
349             }
350             else
351             {
352                 m[i][j] = 0;
353                 m[j][i] = 0;
354             }
355         }
356     }
357
358     return m;
359 }
360
361 //data la matrice trovo i valori di lambda estremeali
362 //per cui in ogni intervallo trovato non cambia il complesso
363 double* find_lambda_value(double** matrix, int n) {
364     int size = n * (n - 1) / 2;
365     double* lambda = (double*)calloc(size, sizeof(double));
366     int i = 0, j = 0, k = 0;
367
368     //mi salvo tutti gli elementi sopra la diagonale della matrice
369     //delle distanze
370     for (i = 0; i < n; i++)
371         for (j = i + 1; j < n; j++) {
372             lambda[k] = matrix[i][j];
373             k++;
374         }
375
376     double l = 0;
377     double app = 0;
378     int iter = 0;
379
380     //ordino le distanze
381     qsort(lambda, size, sizeof(double), compare_double);
382
383     /*printf("lambda\n");
384     for (int i = 0; i < size; i++)
385         printf("%lf ", lambda[i]);
386     */
387
388     //creo un vettore con gli estremi degli intervalli andando a
389     //leggere le distanze
390     while (l < lambda[size - 1] || iter < size) {
391         l += 0.5;
392
393         if (lambda[iter] == app) {
394             lambda[iter] = 0;
395             iter++;
396             l -= 0.5;
397             continue;
398         }
399         else if (l > lambda[iter])
400         {
401             app = lambda[iter];
402             lambda[iter] = l;
403             iter++;
404         }
405     }
406
407     return lambda;
408 }
409
410 //data la matrice e gli intervallo creo la successione di amtrici di
411 //adiacenza
412 matrici_persistenza* create_matrix_persistenza(double** matrix, int n,

```



```

410     double* l) {
411         matrici_persistenza* mp = NULL, * app = NULL, * my_new = NULL;
412         int size = (n*(n-1)/2);
413         double* lambda = find_lambda_value(matrix, n);
414
415         //dentro il vettore ci sono zeri che salto
416         int i = 0;
417         while (lambda[i] == 0)
418             i++;
419
420         //creo quando lambda è non nullo la matrice di adiacenza
421         //e mi salvo l'intervallo
422         //per comodità divido il perimetro e l'ultimo elemento della lista
423         //poichè hanno valori di minimo e massimo predefiniti
424         mp = (matrici_persistenza*)malloc(sizeof(matrici_persistenza));
425         mp->l_min = 0.5;
426         mp->l_max = lambda[i]-0.5;
427         mp->size = n;
428         mp->matrix_d = zero_one_matrix(matrix, n, mp->l_max);
429
430         mp->prev = NULL;
431         mp->next = NULL;
432         app = mp;
433         i++;
434
435         while (i < size)
436         {
437             my_new = (matrici_persistenza*)malloc(sizeof(
438                 matrici_persistenza));
439             my_new->l_min = mp->l_max + 0.5;
440             while (lambda[i] == 0)
441             {
442                 if(i < size)
443                     i++;
444                 else
445                     break;
446             }
447             my_new->l_max = lambda[i] - 0.5;
448             my_new->size = n;
449             my_new->matrix_d = zero_one_matrix(matrix, n, my_new->l_max
450                 );
451
452             my_new->prev = mp;
453             my_new->next = NULL;
454             mp->next = my_new;
455             mp = my_new;
456             i++;
457         }
458
459         my_new = (matrici_persistenza*)malloc(sizeof(matrici_persistenza));
460         my_new->l_min = mp->l_max + 0.5;
461         my_new->l_max = my_new->l_min;
462         my_new->size = n;
463         printf("l_max = %lf\n", my_new->l_min);
464         my_new->matrix_d = zero_one_matrix(matrix, n, my_new->l_max);
465         printf("Matrice n");
466         print_matrix(my_new->matrix_d, n, n);
467         my_new->prev = mp;
468         my_new->next = NULL;
469         mp->next = my_new;
470         mp = my_new;
471         l[0] = mp->l_max;
472
473         return app;
474     }
475
476     //funzione per comparare due double usata successivamente
477     int compare_double(const void* a, const void* b) {

```

```

477         double da = *(const double*)a;
478         double db = *(const double*)b;
479
480         if (da < db) return -1;
481         if (da > db) return 1;
482         return 0;
483     }
484
485 #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
17         destra
18         e poi procedendo verso sinistra.
19       * Per ciascun elemento non nullo, l'intervallo è [birth, death] dove:
20         - birth = numero della riga (1-indexed);
21         - death = numero della colonna (1-indexed).
22     - Ogni intervallo viene disegnato come segmento orizzontale e posizionato
23       sulla "prima quota libera"
24       (cioè, sul livello y più basso in cui non si sovrappone ad un altro
25       intervallo).
26
27     """
28     n = len(matrix) # dimensione della matrice
29     intervals = [] # lista di intervalli: ciascuno è una tupla (birth, death)
30
31     # Scorriamo le righe (i indici partono da 0; aggiungiamo 1 per ottenere la 1-
32     # indexazione)
33     for i, row in enumerate(matrix):
34         if any(val != 0 for val in row):
35             # Troviamo gli indici degli elementi non nulli e li ordiniamo
36             # decrescentemente (da destra a sinistra)
37             nonzero_indices = [j for j, val in enumerate(row) if val != 0]
38             nonzero_indices.sort(reverse=True)
39
40             birth = lambda_min + i*h # riga corrente in 1-indexing
41             for j in nonzero_indices:
42                 death = lambda_min + j*h # colonna in 1-indexing
43                 num_segments = row[j]
44                 # Aggiungiamo tante istanze quanto il valore in cella
45                 for _ in range(num_segments):
46                     intervals.append((birth, death))
47
48     # Ordiniamo gli intervalli per birth (e per death in ordine decrescente se la
49     # riga è la stessa)
50     intervals.sort(key=lambda x: (x[0], -x[1]))
51
52     # 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**:

$$\langle S \rangle := \bigcap_{S \subset I} I, \quad I \text{ ideale}$$

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 $(d) := \langle \{d\} \rangle = \{ad \mid a \in A\}$ si dice **ideale principale**.

Nel corso abbiamo dimostrato che \mathbb{Z} è a ideali principali, ovvero ogni suo ideale è della forma (d) con $d \in \mathbb{Z}$, quindi se si considera un ideale della forma (a, b) esiste un altro numero d unico a meno del segno, che risulta essere il loro **MCD**, tale che $(a, b) = (d)$.

Per determinare il MCD si può applicare l'algoritmo euclideo delle divisioni successive, metodo applicato nel seguente codice.

Listing 10: MCD.h

```
1 #include <stdlib.h>
2 #include <gmp.h>
3
4 void mcd_euclide (mpz_t, mpz_t, mpz_t);
5 void mcd_binario (mpz_t, mpz_t, mpz_t);
6 void mcd_euclide_array (mpz_t, mpz_t*, int);
7 void mcd_binario_array (mpz_t, mpz_t*, int);
8 void mcd_euclide_concat (mpz_t, mpz_t*, int);
9 void mcd_binario_concat (mpz_t, mpz_t*, int);
10
11 // massimo comun divisore tra n e m, salvato in a, con algoritmo di Euclide. Il
12 // risultato è sempre non negativo.
13 void mcd_euclide (mpz_t a, mpz_t n, mpz_t m) {
14     // creo nuove variabili per non modificare n e m, e le rendo positive per avere
15     // mcd >= 0
16     mpz_t x, y;
17     mpz_inits(x, y, NULL);
18     mpz_abs(x, n); mpz_abs(y, m);
19
20     while (mpz_cmp_si(y, 0) > 0) { // se y > 0 continuo a iterare
21         mpz_fdiv_r(x, x, y); // x diventa il resto di x/y
22         mpz_swap(x, y);
23     }
24     mpz_set(a, x);
25     mpz_clears(x, y, NULL);
26     return;
27 }
```

```

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
            i 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
96 // algoritmo di Euclide esteso. Il risultato è sempre non negativo.
97 void mcd_euclide_array (mpz_t a, mpz_t* integers, int n) {
98     // caso banale: n=1
99     if (n==1) {
100         mpz_set(a,integers[0]);
101         mpz_abs(a,a);
102         return;
103     }
104     // creo nuove variabili per non modificare gli interi dati, e le rendo positive
105     // per avere mcd>=0
106     int i;
107     mpz_t* nums = malloc(n*sizeof(mpz_t));
108     for (i=0; i<n; i++) {
109         mpz_init_set(nums[i],integers[i]);
110         mpz_abs(nums[i],nums[i]);
111     }
112
113     int k=n-1; i=0;
114     // sposto gli zeri in fondo
115     while (i<k) {
116         while (mpz_cmp_si(nums[i],0)>0 && i<n) i++;
117         if (i>=n) break;
118         while (mpz_cmp_si(nums[k],0)==0 && k>=0) k--;
119         if (k<0) break;
120         if (i>=k) break;
121         // ho individuato in posizione i un elemento ==0 e in posizione k un
122         // elemento >0: li scambio
123         mpz_swap(nums[i],nums[k]);
124         i++; k--;
125     }
126     k=n; // k indicherà l'indice dal quale gli elementi sono nulli (se ci sono zeri
127     //): nums[k]==0, nums[k-1]>0. (se non ce ne sono k=n)
128     for (i=0; i<n; i++) {
129         if (mpz_cmp_si(nums[i],0)==0) {
130             k=i;
131             break;
132         }
133     }
134     // caso vettore tutto nullo: restituisco mcd=0
135     if (k==0) {
136         mpz_set_si(a,0);
137         for (i=0; i<n; i++) mpz_clear(nums[i]);
138         return;
139     }
140
141     // ciclo finché ho più di un elemento non nullo
142     while (k>1) {
143         // trovo il minimo elemento non nullo e lo porto in prima posizione
144         int index_min=0;
145         for (i=1; i<k; i++) {
146             if (mpz_cmp(nums[index_min],nums[i])>0) index_min=i;
147         }
148         mpz_swap(nums[index_min],nums[0]);
149
150         // divido per il minimo, calcolo i resti e sposto eventuali zeri ottenuti
151         for (i=1; i<k; i++) {
152             mpz_fdiv_r(nums[i],nums[i],nums[0]); // nums[i] diventa nums[i] mod
153             // nums[0], dove nums[0] è il minimo
154
155             // se dopo la divisione ho ottenuto resto 0, porto il numero in fondo
156             if (mpz_cmp_si(nums[i],0)==0) {
157                 mpz_swap(nums[i],nums[k-1]);
158             }
159         }
160     }
161     // ultimo elemento non nullo è a indice k-1
162     mpz_set(a,nums[k-1]);
163     mpz_clear(a);
164     for (i=0; i<n; i++) mpz_clear(nums[i]);
165 }

```

```

153         k--; i--; // aggiorni 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--; // aggiorni 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;

```



```

280 }
281
282 // massimo comun divisore degli n>0 interi dell'array "integers", salvato in a,
    ottenuto concatenando l'algoritmo binario a coppie
283 void mcd_binario_concat (mpz_t a, mpz_t* integers, int n) {
284     // caso banale: n=1
285     if (n==1) {
286         mpz_set(a, integers[0]);
287         mpz_abs(a, a);
288         return;
289     }
290
291     mcd_binario(a, integers[0], integers[1]);
292     for (int i=2; i<n; i++) {
293         mcd_binario(a, a, integers[i]);
294     }
295
296     return;
297 }

```

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 Bézout**, estremamente utile per determinare gli inversi degli elementi in $(\mathbb{Z}_{(n)})^*$.

Listing 11: Bezout.h

```

1  #include <stdlib.h>
2  #include <gmp.h>
3
4  void bezout (mpz_t, mpz_t, mpz_t, mpz_t, mpz_t);
5  void bezout_array (mpz_t, mpz_t*, mpz_t*, int);
6  int inv_mod (mpz_t, mpz_t, mpz_t);
7  int inv_mod_mcd (mpz_t, mpz_t, mpz_t, mpz_t);
8  void exp_mod (mpz_t, mpz_t, mpz_t, mpz_t);
9  void exp_mod_ui (mpz_t, mpz_t, unsigned int, mpz_t);
10
11 // funzioni ausiliarie per bezout:
12 void init_id_matrix (mpz_t**, int);
13 void swap_rows (mpz_t**, int, int, int);
14 void add_multiple_row (mpz_t**, int, mpz_t, int, int);
15
16 // Calcolo d=mcd(n,m) con algoritmo euclideo (scelgo d>=0), e anche x,y interi tali
    che x*n+y*m=d.
17 void bezout(mpz_t d, mpz_t x, mpz_t y, mpz_t n, mpz_t m) {
18     int i, j;
19     mpz_t a, b, q, temp1, temp2;
20     mpz_inits(a, b, q, temp1, temp2, NULL);
21     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
22     // la matrice M (2*2) conterra' alla fine i coefficienti di bezout nella
        prima riga
23     mpz_t** M = malloc(2*sizeof(mpz_t*));
24     for (i=0; i<2; i++) M[i] = malloc(2*sizeof(mpz_t));
25     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
26
27     // algoritmo di euclide
28     while (mpz_cmp_si(b, 0) > 0) { // se b>0 continuo a iterare
29         mpz_fdiv_qr(q, a, a, b); // a diventa il resto di a/b, q quoziente
30         mpz_swap(a, b);
31
32         // aggiorno matrice dei coefficienti: devo moltiplicare a sinistra
            per [[0, 1]; [1, -q]] (caso particolare di solo due interi)
33         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     }
63     return;
64 }
65
66 // creo nuove variabili per non modificare gli interi dati (positive per
   avere mcd>=0), e alloco la matrice necessaria per calcolare i
   coefficienti
67 int i,j;
68 mpz_t* nums = malloc(n*sizeof(mpz_t));
69 for (i=0; i<n; i++) {
70     mpz_init_set(nums[i],integers[i]);
71     mpz_abs(nums[i],nums[i]);
72 }
73 mpz_t** M=malloc(n*sizeof(mpz_t*));
74 for (i=0; i<n; i++) M[i]=malloc(n*sizeof(mpz_t));
75 init_id_matrix(M,n);
76
77 // algoritmo di euclide:
78 int k=n-1; i=0;
79 // spostato 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 }

```

```

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,
111         // tenendo traccia in M
112         int index_min=0;
113         for (i=1; i<k; i++) {
114             if (mpz_cmp(nums[index_min],nums[i])>0) index_min=i;
115         }
116         mpz_swap(nums[index_min],nums[0]);
117         swap_rows(M,n,index_min,0);
118
119         // divido per il minimo, calcolo i resti e sposto eventuali zeri ottenuti
120         for (i=1; i<k; i++) {
121             mpz_fdiv_qr(q,nums[i],nums[i],nums[0]); // nums[i] diventa nums[i] mod
122             // nums[0], dove nums[0] è il minimo
123             mpz_neg(q,q);
124             add_multiple_row(M,n,q,0,i); // alla riga i tolgo q volte
125             // la prima riga (con indice 0)
126
127             // se dopo la divisione ho ottenuto resto 0, porto il numero in fondo
128             if (mpz_cmp_si(nums[i],0)==0) {
129                 mpz_swap(nums[i],nums[k-1]);
130                 swap_rows(M,n,i,k-1);
131                 k--; i--; // aggiorno gli indici
132             }
133         }
134     }
135
136     // quando esco dal ciclo ho k=1 ossia solo il primo elemento è non nullo:
137     // ho trovato mcd, e nella prima riga di M ho i coefficienti cercati
138     mpz_set(d,nums[0]);
139     for (i=0; i<n; i++) {
140         mpz_set(coeff[i],M[0][i]);
141         if (mpz_cmp_si(integers[i],0)<0) mpz_neg(coeff[i],coeff[i]); //
142         // aggiusto i segni poiché all'inizio avevo reso tutto positivo
143     }
144
145     mpz_clear(q);
146     for (i=0; i<n; i++) mpz_clear(nums[i]);
147     for (i=0; i<n; i++) for (j=0; j<n; j++) mpz_clear(M[i][j]);
148     return;
149 }
150
151 // Calcola in x l'inverso di a modulo n: se esiste restituisce 1, altrimenti 0.
152 // Scelgo x>=0, e suppongo n>0.
153 int inv_mod (mpz_t x, mpz_t a, mpz_t n) {
154     mpz_t d,y;
155     mpz_inits(d,y,NULL);
156
157     bezout(d,x,y,a,n);
158     mpz_fdiv_r(x,x,n); // rendo x tale che 0<x<n
159     if (mpz_cmp_si(d,1)==0) { // caso mcd(a,n)=1: l'inverso esiste ed è x
160         mpz_clears(d,y,NULL);
161         return 1;
162     }
163     // caso mcd(a,n)!=1: l'inverso non esiste
164     mpz_clears(d,y,NULL);
165     return 0;
166 }

```

```

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 multiplico 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         return;
229     }
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 $a_1, a_2, \dots, a_n \in \mathbb{Z}$ tali che $(a_i, a_j) = 1 \ \forall i \neq j$. Allora

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

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

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.h"
6
7  int** input_sys (int**, int);
8  void print_sys (int**, int);
9  int* input_vec (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
19         '):\n");
20     int **sys=NULL;
21     sys=input_sys(sys, m);
22     print_sys(sys, m);
23
24     // vettori necessari per la risoluzione del sistema
25     int *A=NULL;
26     A=input_vec(A, m);
27     int *B=NULL;
28     B=input_vec_one(B, m);
29     int *Alpha=NULL;

```

```

29     Alpha=input_vec(Alpha, m);
30     int *Gamma=NULL;
31     Gamma=input_vec(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 (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
97             }

```

```

98         }
99         Alpha[i]=sys[i][0]; // Alpha è vettore delle alpha
100     }
101
102     int **D=NULL;
103     D=input_null(D, m, 1);
104
105     for (int i=0; i<m; i++) {
106         D[i][0]=B[i]; // inizializziamo D a B
107     }
108
109     int **S=NULL;
110     S=input_id(S, m);
111     int **T=NULL;
112     T=input_id(T, 1);
113     bezout (m, B, D, S, T);
114
115     for (int i=0; i<m; i++) {
116         Gamma[i]=S[0][i]; // Gamma corrisponde ai coefficienti di Bezout,
117         // ossia è la prima riga della matrice S
118     }
119     return;
120 }
121
122 // aggiustiamo la soluzione x rendendola della forma x=a mod b con 0<=a<b
123 int adjust_sol (int x, int mod) {
124     if (abs(x)>=mod) { // se abs(x)>=mod togliamo ad x i multipli di mod perchè
125         a*x=x mod a
126         int temp=0;
127         temp=x/mod;
128         x=x-temp*mod;
129     }
130     if (x<0) { // se x<0 prendiamo la sua classe equivalente positiva
131         x=mod+x;
132     }
133     return x;
134 }

```

2.3 Test di primalità

2.3.1 Test di Fermat

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

$$a^{n-1} \not\equiv 1 \pmod{n}$$

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

```

1 #pragma once
2 #ifndef Fermat
3 #define Fermat
4
5 long int** matNull (long int**, int, int);
6 int* trasfBin (int*, long int, int);
7 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 scriviamo 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 trasformato 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 multiplico c
73                 per la potenza di a corrispondente, sennò non modifco c
74                 c=c*matPot[i][1];
75             }
76             c=adjust_x(c, n); // riporto c congruo modulo n
77         }
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
83             x i multipli di mod perchè a*x=x mod a (abs scritto così perchè abs
84             lavora su int)
85             long int temp=0;
86             temp=x/mod;
87             x=x-temp*mod;
88         }
89         long int ceilModMezzi=(mod/2)+1; // perchè ceil lavora su double
90         if (x>ceilModMezzi) { // prendiamo la classe di equivalente di modulo
91             minore
92             x=x-mod;
93         } else if (x<0 && (-x)>=ceilModMezzi) {
94             x=mod+x;
95         }
96         return x;
97     }
98
99     // modifico in maniera casuale la base a
100     long int change_a (long int a, long int max, long int min) {
101         srand(time(NULL));
102         a=rand()%(max-min+1)+min; // prendiamo una base a scelta casualmente nell'
103         intervallo [2, n-2]
104         return a;
105     }
106
107     // studiamo la primalità dei numeri dispari
108     bool fermat (long int n, int it_max) {
109         long int a=2, max=n-2, min=2;
110         for (int it=1; it<=it_max; it++) {
111             long int d=MCD(a, n); // calcolo il massimo comune divisore tra a e
112             n
113             if (d!=1) { // se è diverso da 1 n non è primo e d è un suo
114                 divisore
115                 printf("%ld non è un numero primo e %ld è un suo divisore\n",
116                     n, d);
117                 return false;
118             } else { // altrimenti verifico se a^(n-1) è congruo o meno a 1 mod
119                 n, in caso affermativo n è uno pseudoprimo e continuo la
120                 verifica con un'altra base a, se non lo è n non è primo, ma in
121                 questo caso non ho informazioni sul suo divisore
122                 long int c=congruenza_a(a, n-1, n);
123                 if (c!=1) {
124                     printf("%ld non è un numero primo\n", n);
125                     return false;
126                 } else {
127                     a=change_a(a, max, min);
128                 }
129             }
130         }
131         return true;
132     }
133
134     long int MCD(long int a, long int b) {
135         long int temp=0;
136         while (b!=0) {
137             temp=b;
138             b=a%b;
139             a=temp;
140         }
141     }

```

```

130     }
131     return a;
132 }
133
134 #endif

```

Listing 14: Fermat.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 fermat (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 (fermat(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 // studiamo la primalità dei numeri dispari: la funzione ritorna true se il numero
43 // è primo, altrimenti ritorna false
44 bool fermat (mpz_t n, mpz_t it_max) {
45     mpz_t a, it, d, c, exp;
46     mpz_inits(a, it, d, c, exp, NULL);
47     gmp_randstate_t state;
48     gmp_randinit_mt(state);
49     gmp_randseed_ui(state, time(NULL));
50     mpz_set_si(it, 0);
51     mpz_set_si(a, 2);
52
53     while (mpz_cmp(it, it_max)<0) { // finchè it < it_max
54         mcd_euclide(d, a, n); // calcolo il massimo comune divisore tra a e
55         n

```

```

54         if (mpz_cmp_si(d, 1)!=0) { // se d!=1, n non è primo e d è un suo
55             divisore
56             gmp_printf("%Zd non è un numero primo e %Zd è un suo
57                 divisore\n", n, d);
58
59             mpz_clears(a, it, d, c, exp, NULL);
60             gmp_randclear(state);
61
62             return false;
63         } else { // altrimenti verifico se a^(n-1) è congruo o meno a 1 mod
64             n, in caso affermativo n è uno pseudoprimo e continuo la
65             verifica con un'altra base a, se non lo è n non è primo, ma in
66             questo caso non ho informazioni sul suo divisore
67             mpz_sub_ui(exp, n, 1); // exp = n - 1
68             exp_mod(c, a, exp, n); // calcoliamo a^(n-1) mod n
69             if (mpz_cmp_si(c, 1)!=0) {
70                 gmp_printf("%Zd non è un numero primo\n", n);
71
72                 mpz_clears(a, it, d, c, exp, NULL);
73                 gmp_randclear(state);
74
75                 return false;
76             } else {
77                 RandNumber(a, n, state);
78                 mpz_add_ui(it, it, 1); // it = it + 1
79             }
80         }
81     }
82
83     // Pulizia della memoria
84     mpz_clears(a, it, d, c, exp, NULL);
85     gmp_randclear(state);
86
87     return true;
88 }
89
90 void RandNumber(mpz_t a, mpz_t n, gmp_randstate_t state) {
91     mpz_t range;
92     mpz_init(range);
93
94     mpz_sub_ui(range, n, 3); // range = (n - 2) - 2 + 1 = n - 3
95
96     // Genera un numero casuale in [0, range]
97     mpz_urandomm(a, state, range);
98     // Sposta il numero generato nell'intervallo [2, n-2]
99     mpz_add_ui(a, a, 2);
100
101     mpz_clear(range);
102 }
103
104 // calcola in x la base b elevata alla potenza exp>=0 modulo n (exp binario)
105 void exp_mod (mpz_t x, mpz_t b, mpz_t exp, mpz_t n) {
106     mpz_t squares, e;
107     mpz_init_set(squares, b);
108     mpz_init_set(e, exp);
109     mpz_set_si(x, 1);
110
111     while (mpz_cmp_si(e, 0)>0) { // finchè e > 0
112         if (mpz_tstbit(e, 0)) { // se bit è 1 multiplico x per il quadrato
113             (mod n), altrimenti lascio così
114             mpz_mul(x, x, squares);
115             mpz_mod(x, x, n);
116         }
117         mpz_mul(squares, squares, squares); // aggiorno il quadrato (mod n)
118         mpz_mod(squares, squares, n); // squares mod n
119         mpz_fdiv_q_2exp(e, e, 1); // shift di 1 i bit di e
120     }
121
122     mpz_clears(squares, e, NULL);

```

```

118         return;
119     }

```

2.3.2 Test di Eulero

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

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

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
53                                 n
54         if (mpz_cmp_si(d, 1)!=0) { // se d!=1, n non è primo e d è un suo
55                                 divisore
56             gmp_printf("%Zd non è un numero primo e %Zd è un suo
57                         divisore\n", n, d);
58
59             mpz_clears(a, it, d, c, exp, n1, NULL);
60             gmp_randclear(state);
61
62             return false;
63         } else { // verifico se a^((n-1)/2) è congruo o meno a più o meno 1
64                 mod n, in caso affermativo n potrebbe essere primo quindi
65                 continuo la verifica con un'altra base a, altrimenti n non è
66                 primo
67                 mpz_sub_ui(exp, n, 1);
68                 mpz_fdiv_q_ui(exp, exp, 2); // exp = (n-1)/2
69                 exp_mod(c, a, exp, n); // calcoliamo a^((n-1)/2) mod n
70                 mpz_sub_ui(n1, n, 1); // la classe -1 corrisponde alla
71                                     classe n-1
72                 if (mpz_cmp_si(c, 1)!=0 && mpz_cmp(c, n1)!=0) {
73                     gmp_printf("%Zd non è un numero primo\n", n);
74
75                     mpz_clears(a, it, d, c, exp, n1, NULL);
76                     gmp_randclear(state);
77
78                     return false;
79                 } else {
80                     RandNumber(a, n, state);
81                     mpz_add_ui(it, it, 1); // it = it + 1
82                 }
83             }
84         }
85
86         // Pulizia della memoria
87         mpz_clears(a, it, d, c, exp, n1, NULL);
88         gmp_randclear(state);
89
90         return true;
91     }
92 }
93
94 void RandNumber(mpz_t a, mpz_t n, gmp_randstate_t state) {
95     mpz_t range;
96     mpz_init(range);
97
98     mpz_sub_ui(range, n, 3); // range = (n - 2) - 2 + 1 = n - 3
99
100    // Genera un numero casuale in [0, range]
101    mpz_urandomm(a, state, range);
102    // Sposta il numero generato nell'intervallo [2, n-2]
103    mpz_add_ui(a, a, 2);
104
105    mpz_clear(range);
106 }
107
108 // calcola in x la base b elevata alla potenza exp>=0 modulo n (exp binario)
109 void exp_mod (mpz_t x, mpz_t b, mpz_t exp, mpz_t n) {
110     mpz_t squares, e;
111     mpz_init_set(squares, b);
112     mpz_init_set(e, exp);
113     mpz_set_si(x, 1);
114
115     while (mpz_cmp_si(e, 0)>0) { // finchè e > 0
116         if (mpz_tstbit(e, 0)) { // se bit è 1 moltiplico x per il quadrato
117                                 (mod n), altrimenti lascio così

```

```

109         mpz_mul(x, x, squares);
110         mpz_mod(x, x, n);
111     }
112     mpz_mul(squares, squares, squares); // aggiorno il quadrato (mod n)
113     mpz_mod(squares, squares, n); // squares mod n
114     mpz_fdiv_q_2exp(e, e, 1); // shift di 1 i bit di e
115 }
116
117 mpz_clears(squares, e, NULL);
118
119 return;
120 }

```

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

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

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 $a \in \{2, \dots, n-1\}$ fa da testimone.

Listing 16: Solovay-Strassen

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <gmp.h>
4
5  // Funzione per calcolare il massimo comune divisore (MCD)
6  void mcd(mpz_t result, mpz_t a, mpz_t b) {
7      mpz_gcd(result, a, b);
8  }
9
10 // Esponenziazione modulare binaria
11 type void moduloExponentiation(mpz_t result, mpz_t base, mpz_t exp, mpz_t mod) {
12     mpz_powm(result, base, exp, mod);
13 }
14
15 // Calcolo del simbolo di Jacobi
16 int jacobiSymbol(mpz_t a, mpz_t n) {
17     return mpz_jacobi(a, n);
18 }
19
20 // Test di Solovay-Strassen
21 int solovayStrassen(mpz_t n, int iterations) {
22     if (mpz_cmp_ui(n, 2) < 0) return 0;
23     if (mpz_cmp_ui(n, 2) == 0) return 1;
24     if (mpz_even_p(n)) return 0;
25
26     gmp_randstate_t state;
27     gmp_randinit_mt(state);
28     gmp_randseed_ui(state, rand());
29
30     mpz_t a, n_minus1, exp, gcd, modExp;
31     mpz_inits(a, n_minus1, exp, gcd, modExp, NULL);
32
33     mpz_sub_ui(n_minus1, n, 1);
34     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

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

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

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <gmp.h>
4
5 // Calcolo MCD con algoritmo di Euclide
6 void mcd(mpz_t result, const mpz_t a, const mpz_t b) {
7     mpz_gcd(result, a, b);
8 }
9
10 // Esponenziazione modulare binaria
11 void moduloExponentiation(mpz_t result, const mpz_t base, const mpz_t exp, const
    mpz_t mod) {
12     mpz_powm(result, base, exp, mod);
13 }
14
15 // Funzione ausiliaria per verificare se n è composto
16 int isComposite(const mpz_t n, const mpz_t a, const mpz_t d, long h) {
17     mpz_t x, prev, temp;
18     mpz_inits(x, prev, temp, NULL);
19
20     //  $x = a^d \bmod n$ 
21     moduloExponentiation(x, a, d, n);
22
23     // Se  $x == 1$  o  $x == n - 1$ , non posso concludere che sia composto
24     mpz_sub_ui(temp, n, 1);
25     if (mpz_cmp_ui(x, 1) == 0 || mpz_cmp(x, temp) == 0) {
26         mpz_clears(x, prev, temp, NULL);
27         return 0;
28     }
29
30     // Itero attraverso le squadrature
31     for (long i = 1; i < h; i++) {
32         mpz_set(prev, x);
33         mpz_mul(x, x, x);
34         mpz_mod(x, x, n);
35
36         // Se  $x == 1$  e  $prev \neq n - 1$ , allora è composto
37         if (mpz_cmp_ui(x, 1) == 0) {
38             if (mpz_cmp(prev, temp) != 0) {
39                 mpz_clears(x, prev, temp, NULL);
40                 return 1;
41             } else {
42                 mpz_clears(x, prev, temp, NULL);
43                 return 0;
44             }
45         }
46
47         // Se  $x == n - 1$ , esco dal ciclo
48         if (mpz_cmp(x, temp) == 0) {
49             mpz_clears(x, prev, temp, NULL);
50             return 0;
51         }
52     }
53
54     // Se dopo tutte le iterazioni  $x \neq 1$ , allora è sicuramente composto
55     if (mpz_cmp_ui(x, 1) != 0) {
56         mpz_clears(x, prev, temp, NULL);
57         return 1;
58     }
59
60     mpz_clears(x, prev, temp, NULL);
61     return 1;
62 }
63
64 // Test di Miller-Rabin
65 int millerRabin(const mpz_t n, int iterations) {
66     if (mpz_cmp_ui(n, 2) < 0) return 0;
67     if (mpz_cmp_ui(n, 2) == 0 || mpz_cmp_ui(n, 3) == 0) return 1;
68     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 $x \not\equiv y \pmod n$ e $x \equiv y \pmod p$. 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

```
1  #include <stdio.h>
2  #include <stdbool.h>
3  #include <gmp.h>
4  #include "../Include/mcd.h"
5
6  bool rho_pollard (mpz_t, mpz_t);
7  void funz_f (mpz_t, mpz_t, mpz_t);
8  void funz_g (mpz_t, mpz_t, mpz_t);
9
10 int main () {
11     mpz_t n, d, r;
12     mpz_inits(n, d, r, NULL);
13     printf("Inserire il numero non primo da fattorizzare: n = ");
14     gmp_scanf("%Zd", n);
15
16     mpz_fdiv_r_ui (r, n, 2);
17     if (mpz_cmp_si(r, 0) == 0) { // se il numero è pari allora 2 è un divisore
18         // proprio
19         gmp_printf("Un divisore di %Zd è d = 2\n", n);
20     } else {
21         if (rho_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 // Fattorizza l'intero positivo n utilizzando l'algoritmo rho di Pollard:
35 // - true = d è un divisore proprio di n
36 // - false = non ha trovato un divisore di n
37 bool rho_pollard (mpz_t d, mpz_t n) {
38     mpz_t x, y, f, g, x_new, it;
39     mpz_inits(x, y, f, g, x_new, it, NULL);
40
41     // Assegniamo i valori iniziali x, y e it
42     mpz_set_si(x, 1); // poniamo x = 1
43     mpz_set_si(y, 1); // poniamo y = 1
44     mpz_set_si(it, 0); // poniamo it = 0
45
46     while (mpz_cmp(it, n) < 0) { // finchè it < it_max
47         // 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
56         // 1 ho trovato un divisore di n
57         return true;
58     }
59
60     mpz_set(x, f); // la nuova x è f(x) calcolato prima
61     mpz_set(y, g); // la nuova y è g(y) calcolato prima
62     mpz_add_ui(it, it, 1); // it = it + 1
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

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

il quale ammette come soluzione un'unica coppia (a, b) .

Si scelgano dunque $a, b \in \{0, \dots, n-1\}$ distinti, allora $(a - b, n) \in \{1, d\}$ con d divisore proprio di n . Bisogna quindi scegliere i due valori affinché risolvano

$$a^2 - b^2 \equiv 0 \pmod{n}$$

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. (INSERIRE CODICI)

2.4.3 Metodo $p-1$ di Pollard

Il metodo si basa su una "scommessa": ci si chiede se $n = p_1^{e_1} \dots p_k^{e_k}$ è **b -liscio**, ovvero se esiste un intero b tale che

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

Se ciò avviene allora si può trovare un divisore proprio di n con il seguente metodo.

Listing 19: Pmeno1 Pollard.c

```

1  #include <stdio.h>
2  #include <stdbool.h>
3  #include <string.h>
4  #include <gmp.h>
5  #include "..\Include\mcd.h"
6
7  bool p1_pollard (mpz_t, mpz_t);
8  void exp_mod (mpz_t, mpz_t, mpz_t, mpz_t);
9
10 int main () {
11     mpz_t n, d, r;
12     mpz_inits(n, d, r, NULL);
13     printf("Inserire il numero non primo da fattorizzare: n = ");
14     gmp_scanf("%Zd", n);
15
16     mpz_fdiv_r_ui (r, n, 2);
17     if (mpz_cmp_si(r, 0) == 0) { // se il numero è pari allora 2 è un divisore
18         gmp_printf("Un divisore di %Zd è d = 2\n", n);
19     } else {
20         if (p1_pollard(d, n)) {
21             gmp_printf("Un divisore di %Zd è d = %Zd\n", n, d);
22         } else {
23             gmp_printf("Non è stato trovato un divisore di %Zd\n", n);
24         }
25     }
26
27     // Pulizia della memoria
28     mpz_clears(n, d, r, NULL);
29
30     return 0;
31 }
32
33 bool p1_pollard (mpz_t d, mpz_t n) {
34     mpz_t b, b0, b1, a, m, y, it;
35     mpz_inits(b, b0, b1, a, m, y, it, NULL);
36     mpz_set_si(a, 2);
37     mpz_set_si(m, 1);
38     //mpz_set_si(b0, 2);
39     //mpz_set_si(b1, 66); // perchè la lista dei primi va fino a 61 (da 67 in
40     // poi non ci sono)
41     mpz_set_si(b, 2);
42     int primes[18] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47,
43                     53, 59, 61};
44
45     mpz_t it_max;
46     mpz_inits(it_max, NULL);
47     mpz_set_si(it_max, 1000);
48
49     while (mpz_cmp(b, n) < 0) { // finchè it < it_max
50         //RandNumber(b, b0, b1);
51         //gmp_printf("b=%Zd\n", b);

```

```

50         for (int i=0; i<18; i++) {
51             if (mpz_cmp_si(b, primes[i])<0) { // appena b > primes[i]
52                 interrompo il for
53                 break;
54             } else { // finchè b < primes[i]
55                 mpz_t e, base;
56                 mpz_inits(e, base, NULL);
57                 mpz_set_si(base, primes[i]); // Converti int in
58                     mpz_t
59                 mpz_pow_ui(e, base, strlen(mpz_get_str(NULL,primes[
60                     i],b))-1);
61                 mpz_mul(m, m, e); // m = m * e
62                 mpz_clears(e, NULL);
63             }
64         }
65         exp_mod(y, a, m, n);
66         mpz_sub_ui(y, y, 1);
67         mcd_euclide(d, y, n); // d = MCD(a^m-1, n)
68         // gmp_printf("d=%Zd\n", d);
69         if (mpz_cmp_si(d, 1)!=0 && (mpz_cmp(d, n)!=0)) {
70             return true;
71         }
72         mpz_add_ui(b, b, 1); // it = it + 1
73     }
74 }
75
76 mpz_clears(b, b0, b1, a, m, y, it, NULL);
77
78 return false;
79 }
80
81 // calcola in x la base b elevata alla potenza exp>=0 modulo n (exp binario)
82 void exp_mod (mpz_t x, mpz_t b, mpz_t exp, mpz_t n) {
83     mpz_t squares, e;
84     mpz_init_set(squares, b);
85     mpz_init_set(e, exp);
86     mpz_set_si(x, 1);
87
88     while (mpz_cmp_si(e, 0)>0) { // finchè e > 0
89         if (mpz_tstbit(e, 0)) { // se bit è 1 multiplico x per il quadrato
90             (mod n), altrimenti lascio così
91             mpz_mul(x, x, squares);
92             mpz_mod(x, x, n);
93         }
94         mpz_mul(squares, squares, squares); // aggiorno il quadrato (mod n)
95         mpz_mod(squares, squares, n); // squares mod n
96         mpz_fdiv_q_2exp(e, e, 1); // shifto di 1 i bit di e
97     }
98     mpz_clears(squares, e, NULL);
99
100     return;
101 }

```

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 , inoltre si può fare uso del logaritmo discreto su campi finiti per alleggerire i conti. ù

Listing 20: Pmenol Lenstra.h

```

1  #include <string.h>
2  #include "mcd.h"
3  #include "Bezout.h"
4
5  // Numero di primi da utilizzare nella fattorizzazione di n: deve essere compreso
6  // tra 1 e 18 (poiché ho salvato solo una lista dei primi 18
7  // numeri primi); A SEGUITO DI TEST SCRIVERÒ QUI VALORI RAGIONEVOLI
8  #define NUM_PRIMES 4
9
10 int lenstra (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, 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
22 // curve ellittiche: se riesce restituisce 1 e salva in
23 // d un divisore proprio di n, altrimenti restituisce 0.
24 int lenstra (mpz_t d, mpz_t n) {
25
26     // salvo una lista di primi da utilizzare nel tentativo di fattorizzare n:
27     // scelgo i primi 18 poiché la funzione mpz_get_str, che uso
28     // per calcolare il logaritmo base p che serve per la stima di Hasse, accetta
29     // come secondo argomento solo numeri da 2 a 62. Teoricamente si
30     // può aumentare, ma bisogna trovare un modo alternativo per calcolare la stima
31     // , e in ogni caso solitamente bastano pochi primi.
32     int primes[18] = {2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61};
33
34     mpz_t r;
35     mpz_init(r);
36     // faccio due prime divisioni test: nell'algoritmo serve n dispari, e sul libro
37     // di Koblitz chiede che i primi che dividano n siano >3.
38     mpz_fdiv_r_ui(r,n,2);
39     if (mpz_cmp_si(r,0)==0) {
40         mpz_set_ui(d,2);
41         mpz_clear(r);
42         return 1;
43     }
44     mpz_fdiv_r_ui(r,n,3);
45     if (mpz_cmp_si(r,0)==0) {
46         mpz_set_ui(d,3);
47         mpz_clear(r);
48         return 1;
49     }
50     mpz_clear(r);
51
52     mpz_t hasse_n, x_P, y_P, x_iter, y_iter, a, b, delta; // x_P, y_P coordinate del punto
53     // P sulla curva ellittica  $y^2 = x^3 + ax + b$ 
54     mpz_inits(hasse_n, x_P, y_P, x_iter, y_iter, a, b, delta, NULL);
55     unsigned int exp_primes[NUM_PRIMES];
56     create_exp_primes(exp_primes, primes, hasse_n); // inizializzo gli esponenti a
57     // cui elevare ciascun primo
58     mpz_clear(hasse_n);
59
60     // inizio l'algoritmo di Lenstra
61
62     // (ciclo su x_iter e y_iter poiché nel corso dell'algoritmo x_P e y_P vengono
63     // modificati)

```

```

55 // ciclo sulla componente x del punto P da 0 a n-1
56 while (mpz_cmp(x_iter,n)<0) {
57     mpz_set_ui(y_iter,1);
58
59     // ciclo sulla componente y del punto P da 1 a n-1 (parte da 1 altrimenti
        il primo raddoppio non è definito)
60 while (mpz_cmp(y_iter,n)<0) {
61     mpz_set_ui(a,0);
62
63     // ciclo sulla scelta di a tra 0 e n-1
64 while (mpz_cmp(a,n)<0) {
65
66         mpz_set(x_P,x_iter);
67         mpz_set(y_P,y_iter);
68
69         calcola_b(b,x_P,y_P,a,n); // calcolo b affinché P sia sulla curva
            ellittica
70         calcola_delta(delta,a,b,n); // calcolo discriminante della curva
            ellittica (mod n)
71
72         if (mpz_cmp_si(delta,0)==0) goto retry; // se delta=0 cambio a e
            riprovo
73
74         mcd_binario(d,delta,n);
75
76         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)
77             return 1;
78         }
79
80         // inizio i calcoli P --> m*P
81
82         int i,j,outcome;
83         for (i=0; i<exp_primes[0]; i++) { // calcolo P--> 2^(e_2)*P
84
85             outcome=double_elliptic(d,x_P,y_P,x_P,y_P,a,n);
86             if (outcome==-1) goto retry; // se non posso raddoppiare cambio
                a e riprovo
87             if (outcome==0) return 1; // successo: ho trovato in d un
                divisore proprio di n
88         }
89
90         // calcolo le potenze con i primi della lista maggiori di 2
91         for (j=1; j<NUM_PRIMES; j++) {
92             for (i=0; i<exp_primes[j]; i++) { // calcolo P-->k^(e_k)*P, con
                k primo
93
94                 outcome=multiply_elliptic(d,x_P,y_P,x_P,y_P,primes[j],a,n);
95                 if (outcome==-1) goto retry; // se non posso moltiplicare
                    cambio a e riprovo
96                 if (outcome==0) return 1; // successo: ho trovato in d un
                    divisore proprio di n
97             }
98         }
99
100         // se arrivo qui ho calcolato mP senza trovare divisori: cambio a e
            riprovo
101
102         retry: mpz_add_ui(a,a,1); // sommo 1 ad a e riprovo
103     }
104     mpz_add_ui(y_iter,y_iter,1); // sommo 1 a y e riprovo
105 }
106
107     mpz_add_ui(x_iter,x_iter,1); // sommo 1 a x e riprovo
108 }
109
110 // se arrivo qui non ho trovato divisori di n per alcun P e alcun a:
111     restituisco 0

```

```

112     mpz_clears(x_P, y_P, x_iter, y_iter, a, b, delta, NULL);
113     return 0;
114 }
115
116 // tenta di calcolare P-->2P sulla curva ellittica; per farlo serve calcolare l'
117 // inverso di 2*y_P mod n, quindi si hanno i casi: se mcd(2y_P,n)=n allora
118 // restituisce -1 (dovrò cambiare curva); se mcd(2y_P,n)=1 il calcolo può essere
119 // portato a termine e restituisco 1; altrimenti ho trovato un divisore
120 // proprio d di n, e restituisco 0.
121 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
122 , mpz_t n) {
123     mpz_t y_P2_inv, temp1;
124     mpz_inits(y_P2_inv, temp1, NULL);
125     mpz_mul_si(temp1, y_P, 2);
126     inv_mod_mcd(d, y_P2_inv, temp1, n);
127     if (mpz_cmp(d, n) == 0) { // fallimento: restituisco -1
128         mpz_clears(y_P2_inv, temp1, NULL);
129         return -1;
130     }
131     if (mpz_cmp_si(d, 1) != 0) { // successo: restituisco 0
132         mpz_clears(y_P2_inv, temp1, NULL);
133         return 0;
134     }
135     // calcolo 2P: vedi Koblitiz per le formule
136     mpz_t temp2, x_P_copy, y_P_copy; // creo copie per non modificare x_P e y_P nei
137     // calcoli
138     mpz_inits(temp2, x_P_copy, y_P_copy, NULL);
139     mpz_set(x_P_copy, x_P); mpz_set(y_P_copy, y_P);
140     mpz_mul(temp1, x_P_copy, x_P_copy);
141     mpz_mul_si(temp1, temp1, 3);
142     mpz_add(temp1, temp1, a);
143     mpz_fdiv_r(temp1, temp1, n);
144     mpz_mul(temp1, temp1, y_P2_inv);
145     mpz_fdiv_r(temp1, temp1, n);
146     mpz_mul(x_2P, temp1, temp1);
147     mpz_fdiv_r(x_2P, x_2P, n);
148     mpz_mul_si(temp2, x_P_copy, 2);
149     mpz_sub(x_2P, x_2P, temp2);
150     mpz_fdiv_r(x_2P, x_2P, n);
151     mpz_sub(temp2, x_P_copy, x_2P);
152     mpz_mul(y_2P, temp1, temp2);
153     mpz_sub(y_2P, y_2P, y_P_copy);
154     mpz_fdiv_r(y_2P, y_2P, n);
155
156     mpz_clears(y_P2_inv, temp1, temp2, x_P_copy, y_P_copy, NULL);
157     return 1;
158 }
159
160 // tenta di calcolare P+Q sulla curva ellittica; per farlo serve calcolare l'
161 // inverso di x_Q-x_P mod n, quindi si hanno i casi: se mcd(x_Q-x_P,n)=n allora
162 // restituisce -1 (dovrò cambiare curva); se mcd(x_Q-x_P,n)=1 il calcolo può essere
163 // portato a termine e restituisco 1; altrimenti ho trovato un divisore
164 // proprio d di n, e restituisco 0.
165 int sum_elliptic (mpz_t d, mpz_t x_sum, mpz_t y_sum, mpz_t x_P, mpz_t y_P, mpz_t
166 x_Q, mpz_t y_Q, mpz_t n) {
167     mpz_t denom, temp1;
168     mpz_inits(denom, temp1, NULL);
169     mpz_sub(temp1, x_Q, x_P);
170     inv_mod_mcd(d, denom, temp1, n);
171     if (mpz_cmp(d, n) == 0) { // fallimento: restituisco -1
172         mpz_clears(denom, temp1, NULL);
173         return -1;
174     }
175     if (mpz_cmp_si(d, 1) != 0) { // successo: restituisco 0
176         mpz_clears(denom, temp1, NULL);
177         return 0;
178     }
179     // calcolo P+Q: vedi Koblitiz per le formule
180     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
175 mpz_inits(temp2,x_P_copy,y_P_copy,x_Q_copy,y_Q_copy,NULL);
176 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);
177 mpz_sub(temp1,y_Q_copy,y_P_copy);
178 mpz_mul(temp1,temp1,denom);
179 mpz_fdiv_r(temp1,temp1,n);
180 mpz_mul(x_sum,temp1,temp1);
181 mpz_sub(x_sum,x_sum,x_P_copy);
182 mpz_sub(x_sum,x_sum,x_Q_copy);
183 mpz_fdiv_r(x_sum,x_sum,n);
184 mpz_sub(temp2,x_P_copy,x_sum);
185 mpz_mul(y_sum,temp1,temp2);
186 mpz_sub(y_sum,y_sum,y_P_copy);
187 mpz_fdiv_r(y_sum,y_sum,n);
188
189 mpz_clears(denom,temp1,temp2,x_P_copy,y_P_copy,x_Q_copy,y_Q_copy,NULL);
190 return 1;
191 }
192
193 // tenta di calcolare P-->kP, con k>2 intero, utilizzando raddoppi ripetuti (
    analogo a esponenziazione binaria). Nel farlo deve utilizzare le formule
194 // 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
195 // restituisco 0; altrimenti porto a termine il calcolo e restituisco 1.
196 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) {
197     int outcome;
198     mpz_t temp1,temp2;
199     mpz_inits(temp1,temp2,NULL);
200     mpz_set(temp1,x_P); mpz_set(temp2,y_P); // temp conterranno man mano le
        coordinate dei raddoppi successivi di P
201     mpz_set_si(x_kP,0); mpz_set_si(y_kP,0);
202
203     while (k>0) {
204         if (k&1==1) { // controllo che l'ultimo bit di k sia un 1; se lo è sommo,
            altrimenti lascio così
205
206             outcome=sum_elliptic(d,x_kP,y_kP,x_kP,y_kP,temp1,temp2,n);
207             if (outcome==-1) return -1; // somma non riuscita: devo cambiare curva
208             if (outcome==0) return 0; // successo: ho trovato il divisore d
209             // altrimenti ho portato a termine la somma con successo, e proseguo i
                calcoli
210         }
211         // aggiorno il raddoppio
212         outcome=double_elliptic(d,temp1,temp2,temp1,temp2,a,n);
213         if (outcome==-1) return -1; // raddoppio non riuscito: devo cambiare curva
214         if (outcome==0) return 0; // successo: ho trovato il divisore d
215         // altrimenti ho portato a termine il raddoppio con successo, e proseguo i
            calcoli
216
217         k>>=1; // shift di 1 i bit di k
218     }
219     // se esco dal ciclo ho portato a termine la moltiplicazione P-->kP con
        successo
220
221     mpz_clears(temp1,temp2,NULL);
222     return 1;
223 }
224
225 // calcola in x la stima di Hasse per n: hasse(n)= (radice_quarta(n) + 1)^2
226 void hasse (mpz_t x, mpz_t n) {
227     mpz_root(x,n,4);
228     mpz_add_ui(x,x,2); // sommo 2 poiché mpz_root restituisce la radice troncata
        alla parte intera
229     mpz_mul(x,x,x);
230
231     return;
232 }
233

```

```

234 // calcola gli esponenti a cui deve essere elevato ciascun primo: exp=parte intera
    di log_(base p) (hasse_n)
235 void create_exp_primes (unsigned int* exponents, int* primes, mpz_t hasse_n) {
236     int i;
237     for (i=0; i<NUM_PRIMES; i++) {
238         // trucco per calcolare il logaritmo: mpz_get_str restituisce una stringa
            contenente la rappresentazione di hasse_n in base
239         // primes[i]; per avere la parte intera del logaritmo base p basta quindi
            calcolare la lunghezza della stringa meno 1
240         exponents[i]=strlen(mpz_get_str(NULL,primes[i],hasse_n))-1;
241     }
242
243     return;
244 }
245
246 // 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
247 void calcola_b (mpz_t b, mpz_t x, mpz_t y, mpz_t a, mpz_t n) {
248     mpz_t temp;
249     mpz_init(temp);
250     mpz_mul(b,y,y);
251     mpz_fdiv_r(b,b,n);
252     exp_mod_ui(temp,x,3,n);
253     mpz_sub(b,b,temp);
254     mpz_mul(temp,a,x);
255     mpz_sub(b,b,temp);
256     mpz_fdiv_r(b,b,n);
257
258     mpz_clear(temp);
259     return;
260 }
261
262 // calcola il discriminante della curva ellittica: delta= 27*a^3 +4*b^2 mod n
263 void calcola_delta (mpz_t delta, mpz_t a, mpz_t b, mpz_t n) {
264     mpz_t temp;
265     mpz_init(temp);
266     mpz_mul_ui(delta,a,3);
267     exp_mod_ui(delta,delta,3,n);
268     mpz_mul_ui(temp,b,2);
269     mpz_mul(temp,temp,temp);
270     mpz_add(delta,delta,temp);
271     mpz_fdiv_r(delta,delta,n);
272
273     mpz_clear(temp);
274     return;
275 }

```

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 $tale che$

$$g^x = h$$

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

Listing 21: 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  $a^j \bmod m$ 
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  $a^x = b \bmod m$ .
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);
31     mpz_sub_ui(order, m, 1);
32
33     // Calcoliamo n = ceil(sqrt(order))
34     double order_d = mpz_get_d(order);
35     double n_d = sqrt(order_d);
36     unsigned long n = (unsigned long)ceil(n_d);
37
38     // Allocazione dell'array per i baby step (di dimensione n)
39     Pair* baby = malloc(n * sizeof(Pair));
40     if (!baby) {
41         fprintf(stderr, "Errore di allocazione della memoria.\n");
42         mpz_clear(order);
43         return 0;
44     }
45
46     // Inizializziamo i baby step: baby[j].value =  $a^j \bmod m$ , per j = 0, ..., n-1.
47     mpz_t cur;
48     mpz_init_set_ui(cur, 1); //  $a^0 \bmod m = 1$ 
49     for (unsigned long j = 0; j < n; j++) {
50         mpz_init(baby[j].value);
51         mpz_set(baby[j].value, cur);
52         baby[j].exponent = j;
53         // cur = cur * a mod m
54         mpz_mul(cur, cur, a);
55         mpz_mod(cur, cur, m);
56     }
57     mpz_clear(cur);
58
59     // Ordiniamo la tabella dei baby step
60     qsort(baby, n, sizeof(Pair), cmp_pair);
61
62     // Calcoliamo  $a^n \bmod m$ 
63     mpz_t a_n;
64     mpz_init(a_n);
65     mpz_powm_ui(a_n, a, n, m);
66
67     // Calcoliamo factor =  $(a^n)^{-1} \bmod m$ 
68     mpz_t factor;
69     mpz_init(factor);
70     if (mpz_invert(factor, a_n, m) == 0) {
71         fprintf(stderr, "Inversione modulo fallita.\n");
72         for (unsigned long j = 0; j < n; j++) {

```

```

73         mpz_clear(baby[j].value);
74     }
75     free(baby);
76     mpz_clear(a_n);
77     mpz_clear(factor);
78     mpz_clear(order);
79     return 0;
80 }
81 mpz_clear(a_n);
82
83 // Impostiamo gamma = b
84 mpz_t gamma;
85 mpz_init_set(gamma, b);
86
87 // Loop sui giant step: per ogni i = 0,..., n, cerchiamo gamma nei baby step
88 for (unsigned long i = 0; i <= n; i++) {
89     // Prepariamo una chiave temporanea per la ricerca
90     Pair key;
91     mpz_init(key.value);
92     mpz_set(key.value, gamma);
93     key.exponent = 0; // non rilevante
94
95     Pair* found = bsearch(&key, baby, n, sizeof(Pair), cmp_pair);
96     mpz_clear(key.value);
97
98     if (found != NULL) {
99         // Soluzione trovata: x = i * n + found->exponent
100         unsigned long x = i * n + found->exponent;
101         mpz_set_ui(result, x);
102
103         // Liberiamo la memoria allocata per i baby step
104         for (unsigned long j = 0; j < n; j++) {
105             mpz_clear(baby[j].value);
106         }
107         free(baby);
108         mpz_clear(gamma);
109         mpz_clear(factor);
110         mpz_clear(order);
111         return 1;
112     }
113     // Aggiorniamo gamma: gamma = gamma * factor mod m
114     mpz_mul(gamma, gamma, factor);
115     mpz_mod(gamma, gamma, m);
116 }
117
118 // Nessuna soluzione trovata: liberiamo la memoria e restituiamo 0.
119 for (unsigned long j = 0; j < n; j++) {
120     mpz_clear(baby[j].value);
121 }
122 free(baby);
123 mpz_clear(gamma);
124 mpz_clear(factor);
125 mpz_clear(order);
126 return 0;
127 }
128
129 int main(void) {
130     // Inizializziamo le variabili GMP
131     mpz_t a, b, m, result;
132     mpz_inits(a, b, m, result, NULL);
133
134     // Lettura degli input
135     gmp_printf("Inserisci a: ");
136     gmp_scanf("%Zd", a);
137     gmp_printf("Inserisci b: ");
138     gmp_scanf("%Zd", b);
139     gmp_printf("Inserisci m: ");
140     gmp_scanf("%Zd", m);
141
142     // Calcoliamo il logaritmo discreto

```

```

143     if (babyStepGiantStep(result, a, b, m)) {
144         gmp_printf("Soluzione trovata: x = %Zd\n", result);
145     }
146     else {
147         gmp_printf("Nessuna soluzione trovata per l'equazione %Zd^x = %Zd (mod %Zd)
148             .\n", a, b, m);
149     }
150     // Pulizia
151     mpz_clears(a, b, m, result, NULL);
152     return 0;
153 }

```

Listing 22: Log discreto PHS.c

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <gmp.h>
4  #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
33 // primo q
34 // L'algoritmo è una tecnica di ricerca per trovare x tale che:  $g^x = h \pmod{p}$ 
35 // dove g è la base, h è il valore di cui vogliamo trovare il logaritmo, e p è un
36 // modulo primo.
37 // L'algoritmo divide il problema in due parti:
38 // La baby-step (precalcolo) crea una tabella con i valori di  $g^{(j*m)} \pmod{p}$  per j da
39 // 0 a m, dove m è la radice quadrata di q (approssimato).
40 // La giant-step calcola successivamente i valori  $h * g^{(-i*m)} \pmod{p}$  e cerca se uno
41 // di questi è nella tabella.
42 // Se trova una corrispondenza, restituisce la soluzione come la somma degli indici
43 // i e j. Se non trova nulla, restituisce 0.
44 void baby_giant_step(mpz_t result, mpz_t g, mpz_t h, mpz_t p) {
45     mpz_t m, val, g_m, inv_g_m, temp;
46     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  $g^{-m} \bmod p$ 
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
86 // di primi piccoli
87 // Questa funzione sfrutta la fattorizzazione del modulo p come un prodotto di
88 // potenze di primi.
89 // La funzione esegue i seguenti passi:
90 // Calcola  $q^e$  (dove q è un primo e e è l'esponente corrispondente alla potenza di
91 // q).
92 // Calcola i valori di  $g_i$  e  $h_i$  come potenze di g e h modulo p, utilizzando  $q^e$ .
93 // Risolve il logaritmo discreto di  $h_i$  rispetto a  $g_i$  modulo  $q^e$  usando l'
94 // algoritmo di Baby-Step Giant-Step.
95 // Restituisce il risultato parziale del logaritmo discreto modulo  $q^e$ .
96 // Funzione per Pohlig-Hellman per un fattore  $q^e$ 
97 void pohlig_hellman(mpz_t result, mpz_t g, mpz_t h, mpz_t p, mpz_t q, int exp) {
98     mpz_t q_e, exponent, g_i, h_i;
99     mpz_inits(q_e, exponent, g_i, h_i, NULL);
100
101     //  $q_e = q^{\text{exp}}$ 
102     mpz_pow_ui(q_e, q, exp);
103     // Calcola l'esponente:  $(p-1) / q^{\text{exp}}$ 
104     mpz_sub_ui(exponent, p, 1);
105     mpz_divexact(exponent, exponent, q_e);
106     //  $g_i = g^{(p-1)/q^{\text{exp}}} \bmod p$ ,  $h_i = h^{(p-1)/q^{\text{exp}}} \bmod p$ 
107     mod_exp(g_i, g, exponent, p);
108     mod_exp(h_i, h, exponent, p);
109
110     baby_giant_step(result, g_i, h_i, p);

```

```

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
113 // diversi, ottenuti tramite il metodo di Pohlig-Hellman.
114 // Il teorema permette di risolvere un sistema di congruenze, restituendo un
115 // risultato finale modulo M, dove M è il prodotto di tutti i moduli.
116 // La formula del CRT è la seguente:
117 //  $x = \sum_i \{x_i \cdot M_i \cdot \text{inv}(M_i)\} \pmod{M}$ 
118 // dove  $M_i$  è il prodotto dei moduli escluso l'i-esimo, e  $\text{inv}(M_i)$  è l'inverso di
119 //  $M_i$  modulo il modulo corrente.
120 void chinese_remainder(mpz_t result, mpz_t *x, mpz_t *m, size_t len) {
121     mpz_t M, Mi, inv, sum;
122     mpz_init_set_ui(M, 1);
123     for (size_t i = 0; i < len; i++) {
124         mpz_mul(M, M, m[i]);
125     }
126     mpz_init_set_ui(sum, 0);
127     for (size_t i = 0; i < len; i++) {
128         mpz_init(Mi);
129         mpz_divexact(Mi, M, m[i]);
130         mpz_init(inv);
131         mod_inverse(inv, Mi, m[i]);
132         mpz_mul(Mi, Mi, inv);
133         mpz_mul(Mi, Mi, x[i]);
134         mpz_add(sum, sum, Mi);
135         mpz_clear(Mi);
136         mpz_clear(inv);
137     }
138     mpz_mod(result, sum, M);
139     mpz_clears(M, sum, NULL);
140 }
141
142 // Algoritmo di Pohlig-Hellman completo
143 // Funzione principale per il logaritmo discreto
144 // utilizza tutte le precedenti per risolvere il logaritmo discreto in modo
145 // efficiente.
146 // Per ogni primo q_i e il corrispondente esponente e_i, calcola il logaritmo
147 // discreto modulo q_i^e_i utilizzando il metodo di Pohlig-Hellman.
148 // Combina i risultati ottenuti usando il Teorema Cinese del Resto.
149 void pohlig_hellman_algorithm(mpz_t result, mpz_t g, mpz_t h, mpz_t p, mpz_t *q,
150     int *e, size_t len) {
151     mpz_t *x = malloc(len * sizeof(mpz_t));
152     mpz_t *moduli = malloc(len * sizeof(mpz_t));
153
154     for (size_t i = 0; i < len; i++) {
155         mpz_init(moduli[i]);
156         mpz_pow_ui(moduli[i], q[i], e[i]); // moduli[i] = q[i]^e[i]
157         mpz_init(x[i]);
158         pohlig_hellman(x[i], g, h, p, q[i], e[i]);
159     }
160
161     chinese_remainder(result, x, moduli, len);
162
163     for (size_t i = 0; i < len; i++) {
164         mpz_clear(x[i]);
165         mpz_clear(moduli[i]);
166     }
167     free(x);
168     free(moduli);
169 }
170
171 // Main
172 // Legge i valori di g (base), h (target), e p (modulo).
173 // Fa inserire all'utente la fattorizzazione di p, con una lista di primi q con i
174 // loro esponenti e .
175 // 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           ");
187     scanf("%d", &num_factors);
188
189     // Allochiamo dinamicamente gli array per i fattori e gli esponenti
190     mpz_t *q = malloc(num_factors * sizeof(mpz_t));
191     int *exp_array = malloc(num_factors * sizeof(int));
192
193     for (int i = 0; i < num_factors; i++) {
194         mpz_init(q[i]);
195         printf("Fattore primo #%d: ", i + 1);
196         gmp_scanf("%Zd", q[i]);
197         printf("Esponente per questo fattore: ");
198         scanf("%d", &exp_array[i]);
199     }
200
201     pohlig_hellman_algorithm(result, g, h, p, q, exp_array, num_factors);
202     gmp_printf("Logaritmo discreto x trovato: %Zd\n", result);
203
204     // Pulizia della memoria GMP
205     mpz_clear(order);
206     mpz_clears(g, h, p, result, NULL);
207     for (int i = 0; i < num_factors; i++) {
208         mpz_clear(q[i]);
209     }
210     free(q);
211     free(exp_array);
212
213     return 0;
214 }

```

Listing 23: Log discreto Rho Pollard.c

```

1 #include <stdio.h>
2 #include <stdbool.h>
3 #include <gmp.h>
4 #include "../Include/mcd.h"
5 #include "../Include/Bezout.h"
6
7 void rho_pollard (mpz_t, mpz_t, mpz_t, mpz_t);
8 void funz (mpz_t, mpz_t, mpz_t, mpz_t, mpz_t, mpz_t);
9
10 int main() {
11     mpz_t x, g, h, p;
12     mpz_inits(x, g, h, p, NULL);
13     printf("Risolvere l'equazione g^x=h nel gruppo (Z/(p))* (p primo) con i
           seguenti dati:\n");

```



```

14     printf("g = ");
15     gmp_scanf("%Zd", g);
16     printf("h = ");
17     gmp_scanf("%Zd", h);
18     printf("p = ");
19     gmp_scanf("%Zd", p);
20
21     rho_pollard(x, g, h, p);
22     gmp_printf("La soluzione del log discreto è x = %Zd\n", x);
23
24     mpz_clears(x, g, h, p, NULL);
25     return 0;
26 }
27
28 void rho_pollard (mpz_t x, mpz_t g, mpz_t h, mpz_t p) {
29     mpz_t gamma, a, b, lambda, alpha, beta, p1, temp_a, temp_b, invb, d, c,
30         invc, pd, ad, temp, it, it_max;
31     mpz_inits(gamma, a, b, lambda, alpha, beta, p1, temp_a, temp_b, invb, d, c,
32         invc, pd, ad, temp, it, it_max, NULL);
33
34     mpz_set_si(gamma, 1); // poniamo gamma = 1
35     mpz_set_si(a, 0); // poniamo a = 0
36     mpz_set_si(b, 0); // poniamo b = 0
37
38     mpz_set_si(lambda, 1); // poniamo lambda = 1
39     mpz_set_si(alpha, 0); // poniamo alpha = 0
40     mpz_set_si(beta, 0); // poniamo beta = 0
41
42     mpz_sub_ui(p1, p, 1); // p1 = p - 1;
43     mpz_set_si(it, 0);
44     mpz_set_si(it_max, 1000);
45
46     bool test=false;
47
48     while (mpz_cmp(it, it_max)<0) {
49         funz(gamma, a, b, p, p1, g, h);
50         funz(lambda, alpha, beta, p, p1, g, h);
51         funz(lambda, alpha, beta, p, p1, g, h); // si muove al doppio della
52             velocità
53
54         if (mpz_cmp(gamma, lambda)==0) {
55             mpz_sub(temp_b, beta, b);
56             mpz_sub(temp_a, a, alpha);
57             mcd_euclide(d, temp_b, p1);
58             if (mpz_cmp_si(d, 1)==0) {
59                 inv_mod(invb, temp_b, p1); // calcolo l'inverso di
60                     temp_b mod p1
61                 mpz_mul(x, invb, temp_a);
62                 mpz_mod(x, x, p1);
63                 break;
64             } else {
65                 mpz_fdiv_q(pd, p1, d); // calcolo p2 = p1 / d
66                 mpz_fdiv_q(c, temp_b, d);
67                 mpz_fdiv_q(ad, temp_a, d);
68                 inv_mod(invc, c, pd); // calcolo l'inverso di c mod
69                     p2
70                 mpz_mul(x, invc, ad);
71                 mpz_mod(x, x, pd); // unica soluzione mod p1/d (
72                     ossia pd)
73
74                 // studio a mano le d soluzioni mod p1
75                 mpz_t i;
76                 mpz_inits(i, NULL);
77                 mpz_set_si(i, 0);
78                 while (mpz_cmp(i, d)<0) {
79                     if (mpz_cmp_si(i, 0)!=0) {
80                         mpz_add(x, x, d); // x = x + d
81                     }
82                     mpz_powm(temp, g, x, p);
83                     if (mpz_cmp(temp, h)==0) {

```

```

78             test=true;
79             break;
80         }
81         mpz_add_ui(i, i, 1);
82     }
83     mpz_clears(i, NULL);
84 }
85 }
86 if (test==true) {
87     break;
88 }
89
90     mpz_add_ui(it, it, 1);
91 }
92
93     mpz_clears(gamma, a, b, lambda, alpha, beta, p1, temp_a, temp_b, invb, d, c
94     , invc, pd, ad, temp, it, it_max, NULL);
95     return;
96 }
97
98 void funz (mpz_t x, mpz_t a, mpz_t b, mpz_t p, mpz_t p1, mpz_t g, mpz_t h) {
99     mpz_t temp;
100     mpz_inits(temp, NULL);
101     // suddividiamo il campo in 3 sottoinsiemi di uguale cardinalità con le
102     // classi di resto mod 3
103     mpz_fdiv_r_ui(temp, x, 3); // temp = x % 3
104
105     if (mpz_cmp_si(temp, 0)==0) {
106         mpz_mul(x, x, x); // x = x * x
107         mpz_mod(x, x, p); // riporto x mod p
108         mpz_mul_ui(a, a, 2); // a = a * 2
109         mpz_mod(a, a, p1); // riporto a mod p-1
110         mpz_mul_ui(b, b, 2); // b = b * 2
111         mpz_mod(b, b, p1); // riporto b mod p-1
112     } else if (mpz_cmp_si(temp, 1)==0) {
113         mpz_mul(x, x, g); // x = x * g
114         mpz_mod(x, x, p); // riporto x mod p
115         mpz_add_ui(a, a, 1); // a = a + 1
116         mpz_mod(a, a, p1); // riporto a mod p-1
117         // b resta invariato
118     } else {
119         mpz_mul(x, x, h); // x = x * g
120         mpz_mod(x, x, p); // riporto x mod p
121         // a resta invariato;
122         mpz_add_ui(b, b, 1); // a = a + 1
123         mpz_mod(b, b, p1); // riporto a mod p-1
124     }
125
126     mpz_clears(temp, NULL);
127     return;
128 }

```

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 $[y] = [x^e] \bmod n$ 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 \mathbb{Z}_p

Listing 24: Radici Modulo.h

```
1 #include <time.h>
2 #include "Bezout.h"
3
4 int radice_mod (mpz_t, mpz_t, mpz_t);
5
6 // calcolo in x, se esiste, la radice quadrata di a modulo p, con p primo. Se
  // esiste restituisce 1, altrimenti 0.
7 int radice_mod (mpz_t x, mpz_t a, mpz_t p) {
8     mpz_t a_modp;
9     mpz_init(a_modp);
10    mpz_fdiv_r(a_modp,a,p);
11
12    // caso p=2:
13    if (mpz_cmp_si(p,2)==0) {
14        if (mpz_cmp_si(a_modp,0)==0) mpz_set_si(x,0); // se a==0 mod 2 la radice è
          0
15        else mpz_set_si(x,1); // se a==1 mod 2 la radice è 1
16        mpz_clear(a_modp);
17        return 1;
18    }
19    // caso a==0 mod p: la radice è 0
20    if (mpz_cmp_si(a_modp,0)==0) {
21        mpz_set_si(x,0);
22        mpz_clear(a_modp);
23        return 1;
24    }
25
26    // controllo che a sia un quadrato mod p: se il simbolo di jacobi è diverso da
  // 1, a non è un quadrato
27    int j = mpz_jacobi(a_modp,p); // DA SOSTITUIRE CON IL SIMBOLO DI JACOBI SCRITTO
  // DA NOI
28    if (j!=1) {
29        mpz_clear(a_modp);
30        return 0;
31    }
32
33    // per numeri casuali:
34    gmp_randstate_t randstate;
35    gmp_randinit_default(randstate);
36    unsigned long int t = time(NULL);
37    gmp_randseed_ui(randstate, t);
38
39    unsigned long int h;
40    mpz_t d,alpha,beta,a_inv,exp,temp;
41    mpz_inits(d,alpha,beta,a_inv,temp,NULL);
42    inv_mod(a_inv,a,p);
43
44    // scrivo p-1 = 2^h *d, con d dispari
45    mpz_sub_ui(d,p,1);
46    h = mpz_scan1(d,0);
47    mpz_fdiv_q_2exp(d,d,h);
48
49    // cerco un non-quadrato
50    do {
51        mpz_urandomm(beta,randstate,p); // genero un numero casuale tra 0 e p-1
52    } while (mpz_jacobi(beta,p)!=-1); // DA SOSTITUIRE CON IL SIMBOLO DI JACOBI
  // SCRITTO DA NOI
53
54    exp_mod(beta,beta,d,p); // inizializzo beta=(non-quadrato)^d
55    // inizializzo x=a^((d+1)/2), ossia x^2*a^-1=a^d radice 2^(h-1)-esima di 1
56    mpz_add_ui(temp,d,1);
```

```

57     mpz_fdiv_q_2exp(temp,temp,1);
58     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
67     -1)-esima di 1 (dove p-1=2^h *d); voglio arrivare a h=1, ossia
68     con alpha radice 1-esima di 1 ==> alpha=1, da cui x radice di a (la condizione
69     nel ciclo è h>0 e non h>1 poiché ho già diminuito h di 1).
70     Ad ogni iterazione so che alpha è radice 2^(h-1)-esima di 1, e testo se alpha è
71     anche radice 2^(h-2)-esima di 1 (qui exp=2^(h-1)): se lo è
72     proseguo lasciando x e alpha invariati; se invece non lo è, devo correggere x
73     che diventa x*beta, dove beta sarà radice 2^h-esima primitiva
74     di 1, e aggiornare alpha di conseguenza*/
75
76     while (h>0) {
77         // aggiorno h e exp
78         h--;
79         mpz_fdiv_q_2exp(exp,exp,1);
80         exp_mod(temp,alpha,exp,p);
81         // testo su alpha
82         if (mpz_cmp_si(temp,1)!=0) { // test fallito: aggiorno x e alpha
83             mpz_mul(x,x,beta);
84             mpz_fdiv_r(x,x,p);
85
86             mpz_mul(alpha,x,x);
87             mpz_mul(alpha,alpha,a_inv);
88             mpz_fdiv_r(alpha,alpha,p);
89         }
90
91         mpz_mul(beta,beta,beta); // aggiorno beta che diventa il suo quadrato
92         mpz_fdiv_r(beta,beta,p);
93     }
94     // quando esco h=0 ossia alpha=x^2*a^-1 è radice 1-esima di 1, quindi x radice
95     di a
96
97     mpz_clears(a_modp,d,alpha,beta,a_inv,exp,temp,NULL);
98     gmp_randclear(randstate);
99     return 1;
100 }

```

3.2 Sistemi di interi

Listing 25: 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);
83     printf("e' X:\n");
84     print_matrix(X, n, 1);
85
86     return 0;
87 }

```

3.3 Frazioni Continue

Listing 26: Frazioni continue.h

```

1  #pragma one
2  #ifndef Create_Factor
3  #define Create_Factor
4
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <math.h>
8
9  int* cont_frac_th(int, int);
10 int* cont_frac_impl(int, int);
11
12 int* cont_frac_th(int r, int n)
13 {
14     int* a = NULL;
15     long double x = 0, y = 0, z = 0;
16     a = (int*)calloc(n, sizeof(int));
17     if(!a)
18     {
19         printf("Errore di allocazione della memoria");
20         return NULL;
21     }
22
23     x = sqrtl(r);
24
25     int i = 0;
26     while (i < n)
27     {
28         a[i] = floorl(x);
29         y = x - a[i];
30         z = 1 / y;
31         x = z;
32         i++;
33     }
34     return a;
35 }
36
37 int* cont_frac_impl(int r, int n)
38 {
39     int* a = NULL;
40     a = (int*)calloc(n, sizeof(int));
41     if(!a)
42     {
43         printf("Errore di allocazione della memoria");
44         return NULL;
45     }
46
47     long double x = sqrtl(r);
48     a[0] = floorl(x);

```

```

49     int i = 1, beta = -a[0], gamma = 1, app = 0;
50
51     while (i < n)
52     {
53         app = (r - beta * beta) / gamma;
54         gamma = app;
55         app = 0;
56         a[i] = floorl((a[0] - beta) / gamma);
57         app = -(a[i] * gamma + beta);
58         beta = app;
59         app = 0;
60         i++;
61     }
62     return a;
63 }
64
65
66 #endif

```

3.4 Funzioni ausiliarie

Listing 27: Matrix.h

```

1  #pragma once
2  #ifndef Matrix
3      #define Matrix
4
5      #include <stdlib.h>
6      #include <stdio.h>
7      #include <stdbool.h>
8      #include "..\Include\Smith.h"
9
10     // Funzioni per la manipolazione delle matrici
11     int** mul_matrix(int**, int, int, int**, int, int);
12     void print_matrix(int**, int, int);
13     void gauss(float**, int);
14     int det_matrix_triangular(int**, int);
15     int rank_matrix_diag(int**, int, int);
16     int** kernel_base(int**, int, int, int*);
17     int** link2matrix_same_row(int**, int, int, int**, int, int);
18     int rank_matrix(int**, int, int);
19     char* matrix_to_json(int**, int); // Funzione per convertire una matrice in
        una stringa JSON per passarla a python
20
21
22     //Funzione per calcolare la moltiplicazione tra due matrici
23     int** mul_matrix(int** matrix1, int row1, int col1, int** matrix2, int row2
        , int col2) {
24         if (col1 != row2) {
25             return NULL;
26         }
27         int** result = NULL;
28         result = (int**)calloc(row1, sizeof(int*));
29         for (int i = 0; i < row1; i++) {
30             result[i] = (int*)calloc(col2, sizeof(int));
31         }
32         for (int i = 0; i < row1; i++) {
33             for (int j = 0; j < col2; j++) {
34                 for (int k = 0; k < col1; k++) {

```



```

35         result[i][j] += matrix1[i][k] * matrix2[k][
36             j];
37     }
38 }
39 return result;
40 }
41
42 //Funzione per stampare una matrice
43 void print_matrix(int** matrix, int row, int col) {
44     for (int i = 0; i < row; i++) {
45         for (int j = 0; j < col; j++) {
46             printf("%d ", matrix[i][j]);
47         }
48         printf("\n");
49     }
50     return;
51 }
52
53
54
55 //Funzione per calcolare l'inversa di una matrice intera
56 int** invert_matrix_integer(int** matrix, int size) {
57     int** A = NULL;
58     A = input_null(A, size, size);
59     int** inv = NULL;
60     inv = input_null(inv, size, size);
61     if (!A || !inv) {
62         free(A);
63         free(inv);
64         return NULL;
65     }
66
67     // Copia della matrice originale in A e inizializzazione della matrice
68     // identità in inv
69     for (int i = 0; i < size; i++) {
70         for (int j = 0; j < size; j++) {
71             A[i][j] = matrix[i][j];
72             inv[i][j] = (i == j) ? 1 : 0;
73         }
74     }
75
76     // Eliminazione di Gauss-Jordan
77     for (int i = 0; i < size; i++) {
78         // Se il pivot A[i][i] è 0, cerca una riga sottostante da scambiare
79         if (A[i][i] == 0) {
80             int swapRow = i + 1;
81             while (swapRow < size && A[swapRow][i] == 0)
82                 swapRow++;
83             if (swapRow == size) { // Matrice singolare
84                 free(A);
85                 free(inv);
86                 return NULL;
87             }
88             swapRows(A, i, swapRow, size);
89             swapRows(inv, i, swapRow, size);
90         }
91
92         // Per ottenere l'inversa intera, il pivot deve essere 1 o -1
93         if (A[i][i] != 1 && A[i][i] != -1) {
94             // La matrice non è unimodulare: non possiamo ottenere un'inversa
95             // intera
96             free(A);
97             free(inv);
98             return NULL;
99         }
100
101         // Se il pivot è -1, moltiplica l'intera riga per -1 per renderlo 1
102         if (A[i][i] == -1) {

```

```

102         for (int j = 0; j < size; j++) {
103             A[i][j] = -A[i][j];
104             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
112                                     essere più o meno 1
113             for (int j = 0; j < size; j++) {
114                 A[k][j] -= factor * A[i][j];
115                 inv[k][j] -= factor * inv[i][j];
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
159 // righe quindi non viene con le righe in ordine
160 void gauss_rectangular(int** matrice, int row, int col) {
161     int r = 0;
162     for (int i = 0; i < col; i++) {
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
169                     // le righe non nulle

```

```

169         r = k;
170         break;
171     }
172     else
173         return;
174 }
175 }
176 else r = i;
177     // Normalizza la riga pivot
178     int pivot = matrice[r][i];
179     printf("pivot = %d\n", pivot);
180
181     // Eliminazione verso il basso
182     for (int k = 0; k < row; k++) {
183         if (k == r) continue;
184         double coeff = (double)matrice[k][i]/pivot;
185         printf("matrice[%d][%d] = %d\n", k, i, matrice[k][i]);
186         printf("coeff = %lf\n", coeff);
187         for (int j = i; j < col; j++) {
188             matrice[k][j] -= coeff * matrice[r][j];
189         }
190     }
191 }
192 }
193
194 //Funzione per calcolare il determinante di una matrice triangolare
195 int det_matrix_triangular(int** matrice, int n) {
196     int det = 1;
197     for (int i = 0; i < n; i++) {
198         det *= matrice[i][i];
199     }
200     return det;
201 }
202
203 //Funzione per calcolare il rango di una matrice diagonale
204 int rank_matrix_diag(int** matrix, int row, int col) {
205     int r = 0;
206     for (int i = 0; i < my_min(row, col); i++) {
207         if (matrix[i][i] != 0)
208             r++;
209     }
210     return r;
211 }
212
213 //Funzione per trovare una base del kernel di una matrice
214 int** kernel_base(int** M, int r, int c, int *n) {
215     int** D = NULL;
216     int** S = NULL;
217     int** T = NULL;
218
219     D = input_null(D, r, c);
220     S = input_id(S, r);
221     T = input_id(T, c);
222
223     for (int i = 0; i < r; i++) {
224         for (int j = 0; j < c; j++) {
225             D[i][j] = M[i][j];
226         }
227     }
228
229     SmithNormalForm(D, S, T, r, c);
230     int rk = rank_matrix_diag(D, r, c);
231
232     //calcolo la forma normale di smith e trovo il rango tramite la matrice
233     //diagonale
234     //la base sarà data ultime rk colonne di T
235     int** B = NULL;

```

```

238         if (c - rk == 0) {
239             free(D);
240             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
263     , int row2, int col2) {
264     int** result = NULL;
265     if (row1 != row2) {
266         printf("Errore: Le matrici non hanno lo stesso numero di
267             righe.\n");
268         return NULL;
269     }
270     result = input_null(result, row1, col1 + col2);
271     for (int i = 0; i < row1; i++) {
272         for (int j = 0; j < col1; j++) {
273             result[i][j] = matrix1[i][j];
274         }
275         for (int j = 0; j < col2; j++) {
276             result[i][j + col1] = matrix2[i][j];
277         }
278     }
279     return result;
280 }
281
282 //Funzione per calcolare il rango di una matrice tramite smith
283 //portiamo la amtrice in forma diagonale e calcoliamo il rango della matrice
284 //diagonale molto più veloce
285 int rank_matrix(int** M, int r, int c) {
286     int** D = NULL;
287     int** S = NULL;
288     int** T = NULL;
289
290     D = input_null(D, r, c);
291     S = input_id(S, r);
292     T = input_id(T, c);
293
294     for (int i = 0; i < r; i++) {
295         for (int j = 0; j < c; j++) {
296             D[i][j] = M[i][j];
297         }
298     }
299
300     SmithNormalForm(D, S, T, r, c);
301     int rk = rank_matrix_diag(D, r, c);
302
303     free(D);
304     free(S);
305     free(T);

```

```

305         return rk;
306     }
307
308     char* matrix_to_json(int** matrix, int n) {
309         // Calcoliamo una dimensione massima per il buffer.
310         // Per ogni numero, assumiamo al massimo 12 caratteri (inclusi segno, cifra
311             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 28: Fattorizzazione.h

```

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

```