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

double* solve_backward(double**, double*,double*, int);
double* solve_forward(double**, double*,double*, int);

int main()
{
    int ndim, i, j, k, ind_pivot;
    double **A = NULL;
    double *p_tmp, *x, *y, *b, *diag;
    int *pivot = NULL;
    double el_pivot, tmp;

    // Si legge da standard input la dimensione della matrice

    printf("Inserire la dimensione della matrice di Hilbert:\n");
    scanf("%d", &ndim);

    // Si alloca dinamicamente la matrice e il vettore pivot

    A = (double**)malloc(ndim*sizeof(double *));
    pivot = (int*)malloc((ndim - 1)*sizeof(double));
    x = (double*)malloc(ndim*sizeof(double));
    y = (double*)malloc(ndim*sizeof(double));
    b = (double*)malloc(ndim*sizeof(double));
    diag = (double*)malloc(ndim*sizeof(double));


    for(i = 0; i < ndim; i++)
    {
        A[i] = (double*)malloc(ndim*sizeof(double));
    }

    // Si stampa la matrice e si costruisce il vettore b in maniera tale che la soluzione x sia un vettore con
    // tutti gli elementi uguali a 1

    printf("\nLa matrice di Hilbert e':\n\n");
    for(i = 0; i < ndim; i++)
    {
        b[i] = 0;
        for(j = 0; j < ndim; j++)
        {
            A[i][j] = 1.0/(i+j+1);
            b[i] += A[i][j];
            printf("%f ", A[i][j]);
        }
        printf("\n");
    }

    // Si effettua l'eliminazione gaussiana con pivoting parziale

    for(k = 0; k < ndim-1; k++)
    {
        // Si ricerca l'elemento pivot

        el_pivot = fabs(A[k][k]);
        ind_pivot = k;
        for (i = k+1; i < ndim; i++)
            if(  fabs(A[i][k]) > el_pivot )
            {
                el_pivot = fabs(A[i][k]);
                ind_pivot = i;
            }
        pivot[k] = ind_pivot;

        // Si scambiano le righe se necessario

        if(pivot[k] != k)
        {
            p_tmp = A[k];
            A[k] = A[pivot[k]];
            A[pivot[k]] = p_tmp;
        }

        // Si procede con l'algoritmo di eliminazione Gaussiana

        for(i = k+1; i < ndim; i++)
            if( A[k][k] != 0.0 )
            {
                A[i][k] = A[i][k] / A[k][k];       // si salva il modificatore direttamente nella parte triangolare inferiore della matrice
```

```
83                  for(j = k+1; j < ndim; j++)
84                      A[i][j] -= A[i][k] * A[k][j];
85              }
86              else
87              {
88                  printf("\nErrore: la matrice inserita e' singolare\n");
89                  return -1;
90              }
91
92      }
93
94      // Si ristampa la matrice e il vettore pivot dopo la modifica
95
96      printf("\nLa matrice modificata e':\n\n");
97      for(i = 0; i < ndim; i++)
98      {
99          for(j = 0; j < ndim; j++)
100         {
101             printf("%f ", A[i][j]);
102         }
103         printf("\n");
104     }
105
106     printf("\nIl vettore pivot degli scambi e':\n\n");
107     for(i = 0; i < ndim - 1; i++)
108         printf("%d ", pivot[i]);
109
110     /*
111     Si procede con la risoluzione del sistema lineare Ax=b sfruttando la fattorizzazione
    PA=LU
112     che porta a risolvere due sistemi triangolari Ly=Pb e Ux=y
113     */
114
115     // Si salva la diagonale di A, la si imposta uguale a tutti 1 e si permuta b secondo
    le informazioni del vettore pivot
116     for (i = 0; i < ndim; i++)
117         {
118             diag[i] = A[i][i];
119             A[i][i] = 1.0;
120
121             if(i != ndim-1 && pivot[i] != i)
122             {
123                 tmp = b[i];
124                 b[i] = b[pivot[i]];
125                 b[pivot[i]] = tmp;
126             }
127         }
128
129     y = solve_forward(A, b, y, ndim);
130
131     // Si reimposta la diagonale di A
132
133     for (i = 0; i < ndim; i++)
134         {
135             A[i][i] = diag[i];
136         }
137     x = solve_backward(A, y, x, ndim);
138
139     // Si stampa il vettore soluzione x (come vettore riga)
140
141     printf("\nLa soluzione del sistema e' (rappresentata come vettore riga):\n");
142     for (i = 0; i < ndim; i++)
143         {
144             printf("%f ", x[i]);
145         }
146
147     free(A);
148     free(x);
149     free(y);
150     free(b);
151     free(diag);
152     free(pivot);
153
154     return 0;
155 }
156
157 double* solve_backward(double** U, double* b, double* x, int n)
158 {
159     /*
160     Risolve il sistema Ux=b con U matrice n x n triangolare superiore non singolare
161     usando la backward substitution.
162     */
163
164     int i,j;
```

```
165         double tmp;
166 
167         x[n-1] = b[n-1]/U[n-1][n-1];
168         for(i = n-2; i >= 0; i--)
169         {
170             tmp = 0.0;
171             for(j = i+1; j < n ; j++)
172                 tmp += U[i][j]*x[j];
173 
174             x[i] = (b[i] - tmp)/U[i][i];
175         }
176         return x;
177     }
178 
179     double* solve_forward(double** L, double* b, double* x, int n)
180     {
181         /*
182         Risolve il sistema Lx=b con L matrice n x n triangolare inferiore non singolare
183         usando la forward substitution.
184         */
185 
186         int i,j;
187         double tmp;
188 
189         x[0] = b[0]/L[0][0];
190         for(i = 1; i < n; i++)
191         {
192             tmp = 0.0;
193             for(j = 0; j < i ;j++)
194                 tmp += L[i][j]*x[j];
195 
196             x[i] = (b[i] - tmp)/L[i][i];
197         }
198         return x;
199     }
200 
```