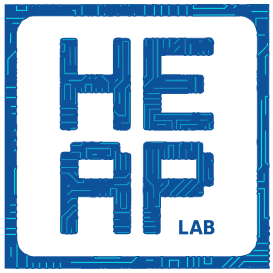
Esercizi su matrici in C

Stefano Cherubin*



02/11/2017

[**Informatica A**] Esercitazione #8

corso per Ing. Gestionale a.a. 2017/18

*<nome>.<cognome>@polimi.it

Indice

1 Simmetria rispetto alla diagonale secondaria

Leggere una matrice quadrata di numeri reali dallo standard input, determinare se si tratta di una matrice "simmetrica rispetto alla diagonale secondaria" e in caso contrario annullare le diagonali principali della matrice inserita e stampare a video la matrice risultato.

1.1 Soluzione C

Listato 1: Simmetria diagonale secondaria

```
1  #include <stdio.h>
2  #define MAXLEN 30
3
4  int main( ) {
5    int main( ) {
6    float mat[MAXLEN][MAXLEN];
7    int i, j, n;
8    int flag;
9
10   /* acquisizione dati*/
11   do {
12     printf("\nOrdine n della matrice");
13     printf(" (0 < n <= MAXLEN) : ");
14     scanf("%d", &n);
15   } while ( n <= 0 || n > MAXLEN);
16
17   printf("\nInserire coefficienti della matrice di
       ordine %d \n", n);
18   /* Lettura della matrice */
19   for (i = 0; i < n; i++)
20     for (j = 0; j < n; j++) {
21       printf("\n mat[%d][%d] := ", i, j);
22       scanf("%f", &mat[i][j]);
23     }
24
25   /* Verifica simmetria rispetto alla diagonale
       secondaria */
26   flag = 1;
27   for (i = n - 1; i > 0 && flag == 1 ; --i)
28     for (j = n - 1; j > n - 1 - i && flag == 1; --j)
29       if (mat[i][j] != mat[n-1-j][n-1-i])
30         flag = 0;
31
32   if (flag == 0) {
33     printf("\nLa matrice non e' simmetrica!");
34     printf("\nAnnullo le diagonali principali....\n");
35     for (i = 0; i < n; i++) {
36       mat[i][i] = 0;
37       mat[i][n - i - 1] = 0;
38     }
39
```

```
40      printf("\n i valori nella matrice sono:\n\n");
41      for (i = 0; i < n ; i++) {
42        for (j = 0; j < n ; j++)
43          printf("%f", mat[i][j]);
44      }
45      printf("\n");
46
47    } else {
48      printf("\nLa matrice inserita e' simmetrica!");
49    }
50    return 0;
51  }
```

2 Quadrato magico

Si progetti e codifichi una funzione C che avendo come parametri d'ingresso una matrice di interi e l'ordine di tale matrice, riempia le celle della matrice con i valori corrispondenti di un quadrato magico di dimensione n, con n dispari.

Un quadrato magico di ordine n contiene i primi n numeri naturali $\left(1, 2, 3, \ldots, n^2\right)$ disposti in modo tale che la somma dei numeri su ogni riga, su ogni colonna e sulle due diagonali principali sia sempre la stessa.

Esiste una regola molto semplice per percorrere la matrice disponendo i numeri interi in ordine crescente.

Partendo col posizionare un 1 nella posizione centrale sull'ultima riga, si percorre la matrice incrementando di una unità il numero di riga e il numero di colonna dell'elemento attuale, avendo cura di considerare i bordi opposti della matrice come adiacenti.

- Se durante questa operazione si individua una cella vuota si scrive il numero con valore successivo a quello della cella di partenza;
- altrimenti, il numero successivo, viene posizionato nella cella avente riga immediatamente precedente a quella della cella di partenza.

Es: $n = 3$

0	0	0	0	0	2	0	0	2	4	0	2
0	0	0	0	0	0	3	0	0	3	0	0
0	1	0	0	1	0	0	1	0	0	1	0

4	0	2	4	0	2	4	0	2	4	0	2
3	5	0	3	5	0	3	5	7	3	5	7
0	1	0	0	1	6	0	1	6	8	1	6

4	9	2
3	5	7
8	1	6

2.1 Soluzione C

Listato 2: Quadrato magico

```
1  #include <stdio.h>
2
3  #define MAX_DIM 51
4
5  void quadratoMagico(int mat[][MAX_DIM], int n);
6
7  int main( ) {
8    int matrix[MAX_DIM][MAX_DIM];
9    int i, j, lim, sum;
10
11   do {
12     printf("\nDim. quadro (dispari <= %d):", MAX_DIM);
13     scanf("%d", &lim);
14   } while (lim > MAX_DIM || lim % 2 == 0);
15
16   quadratoMagico(matrix, lim);
17
18   sum = 0;
19   for (j = 0; j < lim; j++)
20     sum += matrix[0][j];
21
22   printf("\nIl quadrato magico ordine %d e':\n", lim);
23   printf("\nLa somma su ogni linea e' %d.\n", sum);
24
25   for (i = 0; i < lim; i++) {
26     printf("\n");
27     for (j = 0; j < lim; j++)
28       printf("%4d", matrix[i][j]);
29   }
30
31   return 0;
32 } /* end main */
33
34 void quadratoMagico(int mat[][MAX_DIM], int n){
35   int i, j, k;
36
37   for (i = 0; i < n; i++)
38     for (j = 0; j < n; j++)
39       mat[i][j] = 0;
40
41   i = n - 1;
42   j = n / 2;
```

```
43    for (k = 0; k < n * n; k++) {
44      mat[i][j] = k + 1;
45
46  /* indice + 1 % n consente di restare sempre nel
       limite, passando alla riga o colonna successiva
       considerando adiacenti la l'indice n-1 e 0. Infatti
        (n - 1) + 1 % n == 0 */
47      i = (i + 1) % n;
48      j = (j + 1) % n;
49
50      /* Se le nuove coordinate così trovate sono già
           occupate, passa alla riga precedente rispetto
           all'ultima scritta */
51      if (mat[i][j] != 0) {
52        i = i - 2;
53        j = j - 1;
54
55        /* evita indici negativi conservando adiacenza
             tra primo e ultmo */
56        if (i < 0) {
57          i = i + n;
58        }
59        if (j < 0) {
60          j = j + n;
61        }
62      }
63    }
64  } /* end quadratoMagico */
```

3 La guerra contro i giganti di ghiaccio

Odino promise non ci sarebbero stati più giganti di ghiaccio sotto il suo regno. Immaginate di essere suo figlio Thor e cercate di non disattendere le sue aspettative.

Si scriva un programma C che simuli una battaglia contro i giganti di ghiaccio.

Esistono un numero noto di giganti di ghiaccio disposti a caso su una mappa quadrata N x N. Thor può scagliare un fulmine alla volta per incendiare un punto della mappa. Quando un fulmine colpisce la mappa in un punto tutti i giganti di ghiaccio in quel punto si dissolvono e un incendio divampa. Tutti gli altri giganti di ghiaccio si muovono in un punto non incendiato della mappa.

Sembrerebbe tutto molto facile ma Thor ha solo un numero limitato di fulmini che può lanciare. Nel caso in cui non riuscisse a sconfiggere i giganti di ghiaccio, egli subirà l'ira di suo padre, che è ben peggio di perdere una guerra...

3.1 Soluzione C

Listato 3: Thor vs Giganti di ghiaccio

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

#define N 5
#define ICE_GIANTS_POPULATION 10
#define THUNDERS 15
#define BURNING -1

int main() {
  int map[N][N];
  int alive, kill, x, y, turns;
  int i, j;

  /* Map initialization */
  for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
      map[i][j] = 0;
    }
  }

  /* make random really random */
  srand(time(NULL));

  /* Giants population */
  for (i = 0; i < ICE_GIANTS_POPULATION; i++) {
    x = rand() % N;
    y = rand() % N;
    map[x][y] += 1;
  }

  printf("Good morning Thor. Let's end this war!\n");
  turns = THUNDERS;
  alive = ICE_GIANTS_POPULATION;
  while (turns > 0 && alive > 0) {
    do {
      printf("Shoot a thunder to position x y\t");
      scanf(" %d %d", &x, &y);
      if (x < 0 || x >= N || y < 0 || y >= N) {
        printf("Invalid coordinates\n");
      }
    } while (x < 0 || x >= N || y < 0 || y >= N);
```

```
43
44      /* kill them! */
45      kill = map[x][y];
46      if (kill == BURNING) {
47        printf("You hit a burning point\n");
48      } else {
49        alive -= kill;
50        printf("You killed %d giants\n", kill);
51      }
52
53      /* Set the land on fire! */
54      map[x][y] = BURNING;
55
56      /* move other giants */
57      /* reset the position of the survived giants */
58      for (i = 0; i < N; i++) {
59        for (j = 0; j < N; j++) {
60          if (map[i][j] != BURNING) {
61            map[i][j] = 0;
62          }
63        }
64      }
65      /* give them another position */
66      for (i = 0; i < alive; i++) {
67        do {
68          x = rand() % N;
69          y = rand() % N;
70        } while(map[x][y] == BURNING);
71        map[x][y] += 1;
72      }
73
74      turns--;
75    }
76
77    if (alive > 0) {
78      printf("You wasn't able to eliminate them.\n");
79    } else {
80      printf("There are no more frost giants.\n");
81    }
82    return 0;
83  }
```

4 Tappeto elastico

```
#define DIM 10
typedef struct Intv { unsigned int r, c; }
   Casella;
typedef Casella TappetoElastico[DIM][DIM];
```

Pare che tra le discipline degli ottocenteschi "circhi delle pulci" non ci fosse il tappeto elastico (non in quello del Prof. Heckler, almeno). Avrebbe potuto funzionare così:

> la pulce salta sulla prima cella (0,0) del tappeto elastico (quadrato), atterrandovi legge le coordinate (riga e colonna) della prossima cella su cui saltare, e da lì continua a saltare, ogni volta leggendo le coordinate della cella successiva verso cui saltare. Se/quando le coordinate lette indicano un punto esterno al tappeto, la pulce scende (tra gli applausi del pubblico).
>
> Si implementino le seguenti funzioni in C e si spieghi brevemente come funzionano gli algoritmi usati.

... `ciclico(...)` che riceve in input un tappeto elastico e restituisce 1 se esso obbliga a saltare indefinitamente, 0 se invece a un certo punto la pulce potrà scendere

... `contasalti(...)` che riceve un tappeto elastico e misura il numero di salti che la pulce compie prima di scendere (se il tappeto non è ciclico), oppure 1

... `dicoppia(...)` che controlla se un tappeto è adatto all'esibizione di coppia: una seconda pulce inizia a saltare dalla casella (DIM1, DIM1) contemporaneamente alla prima pulce, ed esse continuano a saltare e atterrare in perfetta sincronia fino a uscire assieme dal tappeto, senza mai "scontrarsi", cioè atterrare contemporaneamente sulla stessa casella

4.1 Soluzione C

4.1.1 Ciclico

Viene fatto uso di una struttura di appoggio che simula il tappeto elastico. Inizialmente questa struttura `tec` è tutta *colorata* di 0. Viene *colorata* di 1 quando la corrispondente cella del tappeto elastico viene visitata.

Ad ogni iterazione vengono controllate `jmpOff` e `tec` per verificare di essere atterrati all'interno della scacchiera e su una casella *colorata* di 0.

Listato 4: ciclico

```
1  int ciclico(TappetoElastico te) {
2  /* tappeto aus. per tenere traccia delle caselle
       attraversate */
3    int tec[DIM][DIM] = { 0 };
4  /* flag per controllare se la destinazione del salto è
        fuori dal tappeto: */
5    int jmpOff;
6    jmpOff == 1
7  /* esiste un unico punto d'ingresso al tappeto: (0,0)
       */
8    int r = 0, c = 0;
9    do {
10     tec[r][c] = 1;
11     r = te[r][c].r;
12     c = te[r][c].c;
13     jmpOff = !(r >= 0 && r < DIM && c >= 0 && c < DIM)
           ;
14   } while (jmpOff == 0 && tec[r][c] == 0);
15 /* return 1 se il percorso della pulce è ciclico: cioè
        le coordinate di arrivo del prossimo salto (r,c)
       indicano una posizione già attraversata: tec[r][c]
       == 0  in questo caso il valore di jmpOff è: jmpOff
       == 0
16 return 0, se il percorso della pulce termina: jmpOff
       == 1 */
17   return !JmpOff;
18 }
```

4.1.2 Contasalti

Di questa funzione si propongono diverse varianti. Tali varianti differiscono tra loro per la tecnica di riconoscimento della ciclicità della struttura.

Versione 1 Questa versione esegue nel ciclo con controllo in coda `do { /*code */} while(/*cond*/);` tutti i controlli. Quelli relativi al fatto che il salto resta contenuto nel quadrato e di seguito un controllo sul numero di salti effettuati.

Fissata DIM dimensione della scacchiera, il caso pessimo (in cui la pulce salta su tutte le caselle prima di uscire) prevede di effettuare DIM^2 salti per visitarle tutte (+1 per uscire dalla scacchiera). È quindi matematicamente impossibile essere su un precorso che porta all'uscita dalla scacchiera avendo percorso più di $DIM^2 + 1$ salti.

Listato 5: contasalti (v1)

```
1  /* Contasalti Versione 1 */
2  int contasalti(TappetoElastico te) {
3    int jmpCount = 0, r = 0, c = 0;
4    do {
5      jmpCount++;
6      r = te[r][c].r;
7      c = te[r][c].c;
8    } while(r >= 0 && r < DIM && c >= 0 && c < DIM &&
         jmpCount <= DIM * DIM + 1);
9    /* troppi salti, era un ciclo infinito */
10   if (jmpCount > DIM * DIM + 1)
11     return -1;
12   return 1 + jmpCount;
13 }
```

Versione 2 Sfruttando il codice scritto in precedenza, questa versione dell'algoritmo controlla prima la ciclicità del tappeto e si sviluppa poi supponendo sempre che il tappeto sia aciclico.

Questa variante sarà più lenta perché ripercorre due volte il tappeto (una volta per controllare la ciclicità e una seconda volta per effettuare il conteggio) ma sicuramente ha un corpo più snello rispetto alle altre.

Listato 6: contasalti (v2)

```
1  /* Contasalti Versione 2 */
2  int contasalti(TappetoElastico te) {
3    int jmpCount = 0, r = 0, c = 0;
4    if (ciclico(te) == 1)
5      return 1;
6    do {
7      jmpCount++;
8      r = te[r][c].r;
9      c = te[r][c].c;
10   } while (r >= 0 && r < DIM && c >= 0 && c < DIM);
11   return 1 + jmpCount;
12 }
```

Versione 3 Questa versione prende spunto dalla prima ma utilizza più memoria per migliorare il tempo di esecuzione e la leggibilità del codice. Risulta essere una fusione tra la prima versione e la funzione `ciclico`.

Viene introdotta una flag chiamata `jmpOff` che viene aggiornata ad ogni ciclo per verificare il caso di terminazione per salto fuori dalla scacchiera.

Viene inoltre fatto uso di una struttura di appoggio che simula il tappeto elastico. Inizialmente questa struttura `tec` è tutta *colorata* di 0. Viene *colorata* di 1 quando la corrispondente cella del tappeto elastico viene visitata.

Ad ogni iterazione vengono controllate `jmpOff` e `tec` per verificare di essere atterrati all'interno della scacchiera e su una casella *colorata* di 0.

Listato 7: contasalti (v3)

```
1  /* Contasalti Versione 3 */
2  int contasalti(TappetoElastico te)  {
3    int tec[DIM][DIM] = { 0 };
4    int jmpOff, jmpCount = 0, r = 0, c = 0;
5    do {
6      tec[r][c] = 1;
7      jmpCount++;
8      r = te[r][c].r;
9      c = te[r][c].c;
10     jmpOff = !(r >= 0 && r < DIM && c >= 0 && c < DIM)
           ;
11   } while (jmpOff == 0 && tec[r][c] == 0);
12   if (jmpOff == 1)
13     return 1 + jmpCount;
14   return  1;
15 }
```

4.1.3 DiCoppia

Listato 8: dicoppia

```
1  int dicoppia(TappetoElastico te) {
2    int tec1[DIM][DIM] = { 0 }, tec2[DIM][DIM] = { 0 };
3    int jmpOff1, jmpOff2, r1 = 0, c1 = 0, r9 = DIM-1, c9
         = DIM-1;
4    do {
5      tec1[r1][c1] = 1;
6      tec2[r9][c9] = 1;
7      r1 = te[r1][c1].r;
8      c1 = te[r1][c1].c;
9      r9 = te[r9][c9].r;
10     c9 = te[r9][c9].c;
11     jmpOff1 = !(r1 >= 0 && r1 < DIM && c1 >= 0 && c1 <
           DIM);
12     jmpOff2 = !(r9 >= 0 && r9 < DIM && c9 >= 0 && c9 <
           DIM);
13   } while ( jmpOff1 == 0 && jmpOff2 == 0 && !(r1 == r9
         && c1 == c9) && tec1[r1][c1] == 0 && tec2[r9][c9
        ] == 0 );
14   if (jmpOff1 == 1 && jmpOff2 == 1 && !(r1 == r9 && c1
         == c9))
15     return 1;
16   return 0;
17 }
```

Licenza e crediti

Crediti

Quest'opera contiene elementi tratti da materiale di Gerardo Pelosi redatto per il corso di Fondamenti di Informatica per Ingegneria dell'Automazione a.a. 2014/15.

Licenza beerware[1]

Quest'opera è stata redatta da Stefano Cherubin. Mantenendo questa nota, puoi fare quello che vuoi con quest'opera. Se ci dovessimo incontrare e tu ritenessi che quest'opera lo valga, in cambio puoi offrirmi una birra.

[1] http://people.freebsd.org/~phk/