Svolgimento degli esercizi d'esame del 19-11-05

Esercizio 1: trasformazione da numeri arabi a numeri romani

Un possibile svolgimento può basarsi su una tabella che elenca i possibili valori con loro la traduzione in cifre romane. L'algoritmo di trasformazione diventa molto semplice e si basa sul sottrarre il numero corrente finché è possibile farlo. Bisogna avere l'accortezza di esaminare le cifre in ordine decrescente di valore. Dato che il programma del corso non comprende le struct, realizzo la tabella usando due vettori per contenere i numeri e le loro rispettive traduzioni.

```
#include <string.h>
int cifra[] =
     {1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1 };
char * traduzione[] =
     {"M", "CM", "D", "CD", "C", "XC", "L", "XL", "X", "IX", "V", "IV", "I"};
char romano[10] = "";   // per semplicità uso una stringa allocata staticamente

char * toRomano( int arabo ) {
     // solo i numeri tra 1 e 3999
     if (arabo>3999 || arabo<1)   return romano;
     int i;
     for (i=0 ; i<13 ; i++) {                  // i numeri nel vettore sono 13
          while (arabo>=cifra[i]) {            // finché posso
               arabo -= cifra[i];              // sottraggo la cifra corrente
               strcat(romano, traduzione[i]);  // e ne aggiungo la traduzione
          }
     }
     return romano;
}
```

Esercizio 2: verifica delle mosse degli scacchi

In questo caso conviene dividere la funzione in più funzioni più semplici in modo di poterli anche riusare e tenere nella funzione principale solo i controlli comuni a tutti i tipi di mosse, ovvero:

- la casella di partenza e quella di arrivo debbono essere diverse
- la casella di partenza deve essere piena (non contiene spazio)
- la casella di arrivo deve essere vuota (contiene spazio)

Una volta eseguiti questi tre controlli, i controlli per ciascun pezzo di semplificano:

- torre: o le due x o le due y sono uguali
- alfiere: partenza ed arrivo differiscono dello stesso numero di caselle x ed y (senza segno)
- cavallo: |x-x1|=2 e |y-y1|=1 oppure |x-x1|=1 e |y-y1|=2
- re: |x-x1|<2 e |y-y1|<1
- regina: alfiere && torre
- pedone: |x-x1|=0 e (|y-y1|=1 oppure caso speciale della prima mossa)

Nota: le mosse di pedone, alfiere e torre (e regina) debbono anche controllare che il percorso tra partenza ed arrivo sia libero da altri pezzi

```
#include <stdlib.h>
#define FALSE 0
#define TRUE 1

int mossaBuona( int x, int y, int x1, int y1, char scacchiera[8][8]) {
     if (x<0 || x>7 || x1<0 || x1>7 || y<0 || y>7 || y1<0 || y1>7) return FALSE;
     if (x==x1 && y==y1) return FALSE;
     if (scacchiera[y][x] == ' ' || scacchiera[y1][x1] != ' ') return FALSE;
     switch (scacchiera[y][x]) {
     case 'T':
```

```
    case 't':
        return   verificaTorre(x, y, x1, y1) &&
                 verificaPercorso(x, y, x1, y1, scacchiera);
    case 'A':
    case 'a':
        return   verificaAlfiere(x, y, x1, y1) &&
                 verificaPercorso(x, y, x1, y1, scacchiera);
    case 'P':
        return   verificaPedone(x, y, x1, y1, 6, -1) &&
                 verificaPercorso(x, y, x1, y1, scacchiera);
    case 'p':
        return   verificaPedone(x, y, x1, y1, 2,  1) &&
                 verificaPercorso(x, y, x1, y1, scacchiera);
    case 'Q':
    case 'q':
        return   verificaRegina(x, y, x1, y1) &&
                 verificaPercorso(x, y, x1, y1, scacchiera);
    case 'C':
    case 'c':
        return   verificaCavallo(x, y, x1, y1);
    case 'K':
    case 'k':
        return   verificaRe(x, y, x1, y1);
    default:
        return FALSE;
    }
}
int  verificaTorre( int x, int y, int x1, int y1) {
    return (x==x1 || y==y1);    // mossa solo in orizzontale/verticale
}
int  verificaAlfiere( int x, int y, int x1, int y1) {
    return (abs(x-x1) == abs(y-y1));      // mossa solo in diagonale
}
int  verificaRe( int x, int y, int x1, int y1) {
    return (abs(x-x1)<2 && abs(y-y1)<2);  // mossa di un solo passo
}
int  verificaRegina( int x, int y, int x1, int y1) {
    // regina = alfiere o torre
    return (verificaAlfiere(x, y, x1, y1) || verificaTorre(x, y, x1, y1));
}
int  verificaCavallo( int x, int y, int x1, int y1) {
    // la famosa mossa del cavallo
    return (abs(x-x1)==2 && abs(y-y1)==1 || abs(x-x1)==1 && abs(y-y1)==2);
}
int  verificaPedone( int x, int y, int x1, int y1, int rigastart, int incr) {
    if (x!=x1)         return FALSE; // ci si muove solo avanti/indietro
    if (y1-y == incr)  return TRUE;  // i bianchi in avanti, i neri indietro
    // caso speciale, se è sulla riga di inizio si può muovere di 2 caselle
    if (y == rigastart && (y1-y == 2*incr))
        return TRUE;
    else
        return FALSE;
}
int verificaPercorso(int x, int y, int x1, int y1, char scacchiera[8][8]) {
    int dx = (x<x1 ? 1 : (x==x1 ? 0 : -1));    // incremento in direzione x
    int dy = (y<y1 ? 1 : (y==y1 ? 0 : -1));    // incremento in direzione y
    do {
        x += dx;      // passiamo alla casella successiva
        y += dy;
        if (scacchiera[y][x] != ' ') return FALSE; // casella vuota?
    } while (x!=x1 || y!=y1);    // finché non siamo alla destinazione
    return TRUE;                 // se siamo qui il precorso è vuoto
}
```

Esercizio 3: funzione ricorsiva

La funzione si implementa immediatamente.

Il punto chiave dell'esercizio è mostrare che F non converge per valori negativi dell'argomento.

```
int F(int x) {
     if (x==1) return 42;    // caso base
     if (x % 2 == 0)
          return 2*F(x/2);   // se x è pari
     else
          return 1+F(x+1);   // altrimenti è dispari
}

F(9) = 1+F(10) = 1+(2*F(5)) = 1+(2*(1+F(6))) = 1+(2*(1+(2*F(3))))
     = 1+(2*(1+(2*(1+F(4))))) = 1+(2*(1+(2*(1+(2*F(2))))))
     = 1+(2*(1+(2*(1+(2*(2*F(1))))))) = 1+(2*(1+(2*(1+(2*(2*42))))))
     = 1+(2*(1+(2*(1+(2*84))))) = 1+(2*(1+(2*(1+168)))) = 1+(2*(1+(2*169)))
     = 1+(2*(1+338)) = 1+(2*339) = 1 + 678 = 679
F(-9) = 1+F(-8) = 1+(2*F(-4)) = 1+(2*(2*F(-2))) = 1+(2*(2*(2*F(-1))))
     = 1+(2*(2*(2*(1+F(0))))) = 1+(2*(2*(2*(1+(2*F(0)))))) ....
la funzione a questo punto non termina perché continua a voler calcolare F(0)
```

Esercizio 4: numeri in codifica binaria e in codifica in complemento a 2

Per calcolare quanti bit sono necessari per rappresentare un numero N nella codifica binaria si usa la formula: $B = \lceil \log_2(N+1) \rceil$

(il +1 serve a tener conto correttamente del numero di bit necessari a contenere le potenze di 2)

Quindi per rappresentare i due numei N=96 ed M=224 come numeri binari sono necessari:

- $\lceil \log_2(96+1) \rceil = 8$ infatti $96_{10} = 11000000_2$
- $\lceil \log_2(224+1) \rceil = 9$ infatti $224_{10} = 111000000_2$

La rappresentazione nella codifica in complemento a due di N e M si ottiene aggiungendo un bit 0 per il segno dei numeri positivi. Il numero -M si ottiene dalla rappresentazione di M nella codifica in complemento a due applicando l'operazione di cambio di segno.

- N nella codifica in complemento a due è 011000000
- M nella codifica in complemento a due è 0111000000
- -M so ottiene invertendo M ed aggiungendo 1: 1000111111+1 = 1001000000

La differenza di due numeri nella codifica in complemento a due a 16 bit non è altro che la somma del primo con il complemento del secondo. Prima però bisogna estendere il segno di entrambi i numeri. L'estensione del segno si ottiene copiando il bit più significativo nei bit aggiunti alla sua sinistra.

N-M = 0000000011000000 +

1111111001000000 =

1111111100000000