Informatica - CdL in FISICA

Appello straordinario del 1/6/2012

Scrivere **in stampatello** COGNOME, NOME e MATRICOLA su ogni foglio consegnato

N.B.: Negli esercizi di programmazione, viene valutata anche la leggibilità del codice proposto. Inoltre, non è consentito l'uso di istruzioni che alterino il normale flusso dell'esecuzione (come, ad esempio, `continue`, `break` e istruzioni di `return` all'interno di cicli che ne provochino l'uscita forzata). Infine non è consentito l'uso di variabili statiche.
Laddove è utilizzato, il tipo `boolean` è definito da `typedef enum {false, true} boolean;`

ESERCIZIO 1 (3 punti)
Dato il linguaggio simbolico visto a lezione

```
     LOAD R1 X   |  LOAD R2 X  |  LOAD R1 #C  | SUM R1 R2   |  SUB R1 R2  |
-------------------------------------------------------------------------
     STORE R1 X  |  STORE R2 X |  READ X      |  WRITE X    |  JUMP A     |
-------------------------------------------------------------------------
     JUMPZ A     |  STOP       |
-------------------------------------------------------------------------
```

dove `X` sta per un generico nome simbolico, `A` per un generico indirizzo, e `C` per una generica costante intera tradurre la seguente istruzione:

```
        while (x!=y) x = x+1;
```

Soluzione

```
              LOAD  R1 X
 INIZIO:      LOAD  R2 Y
              SUB   R1 R2
              JUMPZ FINE
              LOAD  R2 X
              LOAD  R1 #1
              SUM   R1 R2
              STORE R1 X
              JUMP INIZIO
FINE:
```

ESERCIZIO 2 (5 punti)
Si dice che una sequenza di interi non negativi è *bilanciata* se contiene lo stesso numero di pari e di dispari. Si scriva una funzione *iterativa* che legga una sequenza di interi non negativi (senza memorizzarla) che termina quando vengono immessi due numeri uguali e che, supponendo di non considerare l'ultimo come parte della sequenza, restituisca:

- 0 se la sequenza non è bilanciata, né ordinata in modo crescente;
- 1 se la sequenza è bilanciata, e non crescente;
- 2 se la sequenza non è bilanciata, ma è crescente;
- 3 la sequenza è bilanciata e crescente.

Soluzione

```
int bilanciataOrdinata() {
  int prec, act;
  int res=0;
  int pari=0, dispari=0;
  boolean ordinata=true;

  scanf("%d", &act);
  if(act%2==0) pari++;
  else dispari++;


  do {
    prec=act;
    scanf("%d", &act);
    if(act!=prec) {
      if(act<prec) ordinata = false;
      if(act%2==0) pari++;
      else dispari++;
    }
  } while(act!=prec);

  if(ordinata) res += 2;
  if(pari==dispari) res++;
  return res;
}
```

ESERCIZIO 3 (6 punti)
Si definisca una funzione *ricorsiva* che dato un array `vet` di interi e la sua dimensione `dim`, restituisca il più piccolo intero maggiore di tutti gli elementi di `vet`. Se ad esempio l'array contiene : `15 100 0 -20`, allora la funzione restituisce `101`.
N.B. Ogni chiamata ricorsiva deve risolvere il problema originario per una porzione dell'array.

```
int minimoMaggiorante(int vet[], int dim) {
  int res, act = v[0]+1;

  if(dim==1) return act;

  res = minimoMaggiorante(vet+1, dim-1);
  if(res>act)
    return res;
  else
    return act;
}
```

ESERCIZIO 4 (16 punti)
Si vuole modellare un gioco con le *carte* mediante una lista concatenata. Una carta e' rappresentata dal suo valore (un intero da 1 a 10) e dal suo seme (Quadri,Cuori, Picche e Fiori). Ogni nodo della lista deve rappresentare una carta con in aggiunta l'informazione di quante carte dello stesso seme seguono nella lista.

(i) (2 punti) Definire i tipi opportuni per rappresentare il gioco.

```
typedef enum { Quadri, Cuori, Picche, Fiori } Seme;

typedef struct carta {
  int val;
  Seme seme;
} Carta;

typedef struct nodo {
  Carta carta;
  int carteSuccessive;
  struct nodo* next;
} NodoMazzo;
typedef NodoMazzo* Mazzo;
```

(ii) (3 punti) Scrivere una funzione *ricorsiva* che data una lista di carte, un valore e il seme di una carta, controlli che tale carta appartenga alla lista *sfruttando al meglio tutte le informazioni contenute nella lista.*

```
boolean appartiene (Mazzo mazzo, int val, Seme seme) {
  if(mazzo==NULL)
    return false;
  else
        {if (mazzo->carta.seme==seme)
           { if  (mazzo->carta.val==val)
               return true;
           else
              {if( mazzo->carteSuccessive==0)
               return false;
              else   return appartiene(mazzo->next, val, seme);}
         }
         else   return appartiene(mazzo->next, val, seme);
          }}
```

(iii) (5 punti) Scrivere una procedura che data una lista di carte, un valore e il seme di una nuova carta, la inserisca prima della prima carta con lo stesso seme, se esiste, altrimenti la inserisca in coda.

```
void inserisci(Mazzo* p_mazzo, int val, Seme seme) {
  Mazzo  aux, nuova, prec;
  boolean done;
  if(p_mazzo!=NULL)  /* sanity check */
{
    nuova = malloc(sizeof(NodoMazzo));
    nuova->carta.val = val;
    nuova->carta.seme = seme;
    nuova->next = NULL;


    if(*p_mazzo==NULL) {
      nuova->carteSuccessive = 0;
      *p_mazzo = nuova;
    }

    else if( *p_mazzo ->carta.seme==seme) {
      nuova->carteSuccessive = *p_mazzo->carteSuccessive + 1;
      nuova->next = *p_mazzo;
```

```
      *p_mazzo = nuova;
    }
    else {
      prec=*p_mazzo;
      aux=*p_mazzo->next;
      done = false;
      while(aux!=NULL && !done) {
        if(aux->carta.seme==seme) {
          prec->next = nuova;
          nuova->next = aux;
          nuova->carteSuccessive = aux->carteSuccessive + 1;
          done = true;
        }
        prec = aux;
        aux = aux->next;
      }
      if(!done) {
        prec->next = nuova;
        nuova->carteSuccessive = 0;
      }
    }
  }
}
```

(iv) (6 punti) Scrivere una funzione *ricorsiva* che data una lista di carte, un valore e il seme di una carta, cancelli la prima occorrenza di tale carta, se esiste, nella lista. La funzione deve restituire *true* se la cancellazione e' avvenuta.

```
boolean cancellaPrima(Mazzo* p_mazzo, int val, Seme seme) {
  Mazzo mazzo;
  if(p_mazzo==NULL) return false; /* sanity check */

  if(*p_mazzo\not=NULL)

  { if(mazzo->carta.seme==seme && mazzo->carta.val==val) {
     mazzo=*p_mazzo;
    *p_mazzo = *p_mazzo->next;
    free(mazzo);
    return true;
  }

  else
     {if(cancellaPrima(&(*p_mazzo->next), val, seme)) {
           if(*p_mazzo->carta.seme==seme) /* aggiorniamo il campo carteSuccessive */
               *p_mazzo->carteSuccessive--;
           return true;}
      else return false;
  } }
  else return false;
}
```