Informatica - CdL in FISICA
PROVA SCRITTA DEL 15/6/2012

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 seguente programma:

```
#include <stdio.h>
int p(int *x, int *y)
{
int ris = *y;
if (*x > 0)
{
*x = *x-1;
*y = *y+2;
ris = p(x,y);
}
return ris;
}
main()
{
int a=A, b=B;
printf("a = %d, b = %d, risultato = %d", A, B, p(&a, &b));
}
```

dove A e B indicano generiche costanti intere, indicare l'output del programma in funzione dei valori di A e B.
Soluzione
Per $A > 0$ la printf stampa

```
a = A, b = B, risultato = B+2A
```

mentre per $A \leq 0$ la printf stampa

```
a = A, b = B, risultato = B
```

Infatti la funzione viene chiamata ricorsivamente finché il primo parametro (a cui é passato originariamente il valore A) diventa minore o uguale a 0. Ad ogni chiamata la funzione diminuisce di 1 il primo parametro e aumenta di 2 con il complessivo risultato di sommare il doppio del valore originale del primo parametro ($2A$) al secondo parametro (B).

Se invece il primo parametro é subito minore o uguale a 0, la funzione ritorna immediatamente il valore del secondo parametro (B).

ESERCIZIO 2 (6 punti)
Leggere una sequenza di caratteri. Scrivere una funzione che dato un carattere c, controlli che la sequenza in input sia palindroma, con elemento centrale esattamente il carattere c. Si assuma che il carattere c appaia una sola volta all'interno della sequenza e in posizione centrale rispetto alla sequenza letta.
N.B. Non si puo' usare allocazione dinamica della memoria.
Soluzione

```
boolean palindroma(char c) {
  char c1,c2;
  boolean ret;

  scanf(" %c", &c1);
  if(c1==c)
    return true;
  else {
```

```
    ret = palindroma(c);
    if(!ret)
      return false;
    else {
      scanf(" %c", &c2);
      return c1==c2;
    }
  }
}
```

ESERCIZIO 3 (5 punti)
Dato un vettore v di elementi di tipo T, la distanza tra due elementi v[i] e v[j], con $i \leq j$ e' data dall'intero non negativo $j - i$. Si definisca una funzione che, dato un vettore di caratteri, la sua dimensione, ed un intero $k > 0$ restituisce true se nel vettore esistono due elementi uguali tra loro e distanti k, e restituisce false altrimenti. Verra' valutata l'efficienza della soluzione proposta: l'algoritmo non deve fare controlli palesemente inutili.
Soluzione

```
boolean elementiUgualiDistantiK(T vet[], int dim, int k) {
  int i;
  boolean trovato = false;

  i=0;
  while( !trovato && (i+k)<dim) {
    if(vet[i]==vet[i+k])
      trovato = true;
    i++;
  }

  return trovato;
}
```

ESERCIZIO 4 (16 punti)
Si vuole modellare una lista che rappresenta i pagamenti dell'IMU di ciascun contribuente ancora dovuti al comune. Si assuma che un contribuente puo' dover pagare l'IMU per una prima casa oppure per una seconda casa oppure per entrambe. Di conseguenza, ogni elemento della lista deve contenere le seguenti informazioni:

- un codice numerico che identifica univocamente il contribuente;
- un campo che indica se si tratta di prima casa oppure no;
- la cifra da pagare.

La lista è mantenuta ordinata per codice del contribuente e, a parità di codice del contribuente, l'elemento relativo al pagamento della prima casa deve precedere quello relativo alla seconda casa.

(2 pt) Definire i tipi opportuni per rappresentare una siffatta lista.

```
typedef struct pagam {
  int contribuente;
  boolean primaCasa;
  int totale;
  struct pagam* next;
} Pagamento;

typedef Pagamento* elencoPagamenti;
```

(4 pt) Data la lista dei pagamenti, scrivere una funzione che conti il numero di contribuenti che devono ancora effettuare almeno un pagamento.

```
int numContribuenti(elencoPagamenti lista) {
  int num = 0;
  int prec = -1;
  while(lista!=NULL) {
    if(prec==-1 || lista->contribuente!=prec) {
      prec = lista->contribuente;
      num++;
    }
    lista = lista->next;
  }
  return num;
}
```

(6 pt) Data la lista dei pagamenti e un'analoga lista di pagamenti già effettuati, scrivere una procedura *ricorsiva* che modifica la lista dei pagamenti, cancellando i pagamenti già effettuati.

```
/*
  Funzione ausiliaria per il confronto fra due elementi:
  ritorna -1 se il primo e' minore, 1 se e' maggiore, 0 se sono uguali.
  Se uno dei due puntatori e' null, quell'elemento e' considerato minore.
  Notare come il segno del controllo del booleano sia invertito: true (che vale 1)
  deve precedere false (che vale 0).
*/
int precede(Pagamento* a, Pagamento* b) {
  if(a==NULL && b==NULL)
    return 0;
  else if(a==NULL && b!=NULL)
    return -1;
  else if(a!=NULL && b==NULL)
    return 1;
  else { /* entrambi non NULL */
    if(a->contribuente < b->contribuente)
      return -1;
    else if(a->contribuente > b->contribuente)
      return 1;
    else if(a->primaCasa > b->primaCasa) /* primaCasa==true deve venire prima  */
      return -1;
    else if(a->primaCasa < b->primaCasa)
      return 1;
    else return 0;
  }
}

void eliminaEffettuati(elencoPagamenti* plista, elencoPagamenti effettuati) {
  elencoPagamenti lista;
  int res;
  if(plista!=NULL) {
    lista = *plista;
    if(lista==NULL || effettuati==NULL)
      return;
    else {
      res = precede(lista, effettuati);
      if(res<0)
        eliminaEffettuati(&(lista->next), effettuati); /* scorri la prima */
      else if(res>0)
        eliminaEffettuati(plista, effettuati->next);  /* scorri la seconda */
      else {
        *plista = lista->next;
```

```
        free(lista);
        eliminaEffettuati(plista, effettuati->next); /* scorri entrambe */
      }
    }
  }
}
```

(4 pt) Data la lista di pagamenti già effettuati e i dati relativi ad un nuovo pagamento (non l'elemento della lista), scrivere una procedura che inserisca un nuovo elemento che contenga i dati forniti, nella posizione dovuta nella lista.

```
void aggiungi(elencoPagamenti* plista, int contribuente, boolean primaCasa, int totale) {
  elencoPagamenti lista, prec;
  Pagamento* nuovo;
  int res;
  if(plista!=NULL) {
    nuovo = malloc(sizeof(Pagamento));
    nuovo->contribuente = contribuente;
    nuovo->primaCasa = primaCasa;
    nuovo->totale = totale;
    nuovo->next = NULL;

    lista = *plista;
    if(lista!=NULL)
      res = precede(nuovo,lista);

    if(lista==NULL || res<0) {
      /* il nuovo va in cima se la lista e' vuota o tutta minore */
      *plista = nuovo;
      nuovo->next = lista;
    }
    else if(res==0)
      free(nuovo); /* se gia' presente, lo ignoriamo */
    else {
      do { /* scorriamo la lista fino al punto di inserimento, o fino in fondo */
        prec = lista;
        if(lista!=NULL) {
          lista = lista->next;
          res = precede(lista,nuovo);
        }
      } while (lista!=NULL && res<0);

      if(res!=0) { /* e agganciamo l'elemento nuovo */
        prec->next = nuovo;
        nuovo->next = lista;
      }
      else
        free(nuovo); /* se gia' presente, lo ignoriamo */
    }
  }
}
```