```
/*
 * Corso di Fondamenti di Informatica
 * Esercizio:
 * tema d'esame
 */

/*
Si consideri un correttore ortografico realizzato in C. Ogni parola Ã¨ rappresentata come un array di esattamente 35
caratteri, ed Ã¨ organizzata come una stringa: i caratteri alfabetici (si usano solo le lettere minuscole) che
effettivamente costituiscono la parola sono riportati allâ€™inizio della sequenza; dopo lâ€™ultimo carattere della parola
viene posto il carattere nullo '\0', mentre le restanti posizioni della sequenza conterranno caratteri non significativi. Il
correttore ortografico dispone di un dizionario della lingua, rappresentato da una sequenza di esattamente 100.000 parole
(non necessariamente in ordine alfabetico) anch'esse rappresentate con caratteri alfabetici minuscoli.
La distanza di battitura tra due lettere Ã¨ definita come il valore assoluto della loro distanza alfabetica. Ad esempio le
lettere "a" e "c" hanno distanza 2, mentre "f" e "a" distanza 5.
La distanza di battitura tra due parole della stessa lunghezza Ã¨ definita come la somma del valore assoluto delle distanze
delle lettere che le compongono, calcolata posizione per posizione. Ad esempio, la distanza tra "casa" e "cane" Ã¨ di
0+0+5+4=9.
La distanza di battitura tra due parole di lunghezza diversa Ã¨ calcolata allo stesso modo di quella tra due parole della
stessa lunghezza,considerando uguale a 26 la distanza di una lettera da una mancante nell'altra parola. Ad esempio, la
distanza tra "test" e "tasto" Ã¨ di 0+4+0+0+26=30.

Parte a. (punti 2)
Si esplicitino le dichiarazioni dei soli prototipi dei seguenti sottoprogrammi:
* Esiste, che riceve in ingresso una parola e un dizionario e restituisce 1 nel caso la parola esista (ovvero sia contenuta)
nel dizionario, 0 altrimenti.
* Distanza, che riceve in ingresso due parole e ne restituisce il valore della distanza di battitura.
* Suggerimento, che riceve una parola e un dizionario e ricerca nel dizionario la parola a distanza di battitura minima.
  La funzione restituisce l'indice della parola in questione all'interno del dizionario.

Parte b. (punti 3)
Si definisca il sottoprogramma Esiste descritto al punto a.

Parte c. (punti 7)
Si definisca il sottoprogramma Distanza descritto al punto a.

Parte d. (punti 5)
Si definisca il sottoprogramma Suggerimento descritto al punto a.
*/
```

```
/*****************************
 Parte a.
*****************************/
#include <stdio.h>
#define MAX_LUNG_PAROLA 36
/* include un carattere aggiuntivo per il terminatore '\0' */
#define NUM_PAROLE 5
/* numero di parole contenute nel dizionario; il valore assegnato e' basso per
consentire il debugging del programma */
/* Nota: per funzionare correttamente, il programma richiede che NUM_PAROLE sia
   almeno pari a 1 */
#define DIST_LETTERA_MANCANTE 26
/* distanza di battitura convenzionale tra una prima lettera qualsiasi e una
   seconda lettera mancante */

typedef char Parola[MAX_LUNG_PAROLA];
typedef Parola Dizionario[NUM_PAROLE];

int Esiste(Parola Par, Dizionario Diz);
int Distanza(Parola Par1, Parola Par2);
int Suggerimento(Parola Par, Dizionario Diz);


/* IMPORTANTE! TUTTO QUANTO Ã^ COMPRESO TRA QUESTA LINEA E QUELLA
   EVIDENZIATA CON @@@@@@@@@@@@@@@@@@@@@@@@@@@@ NON ERA RICHIESTO ALL'ESAME!
   E' STATO INSERITO SOLO PER CONSENTIRE IL DEBUGGING */

#include <string.h>

int main()
{
  Dizionario DizionarioTest;
  /* usato per provare le funzioni richieste dall'esercizio */
  int ElemDiz;
  /* elemento del dizionario correntemente in esame */

  /**** inserisco alcune parole di test in DizionarioTest ***/
  /* N.B. operando in questo modo servono tante strcpy quante sono le parole in
   * un Dizionario; si noti che in C una costante stringa nella forma "..."
   * include giÃ  il terminatore '\0', che dunque non va inserito */
  strcpy(DizionarioTest[0], "aaaaa");
```

```
  strcpy(DizionarioTest[1], "bbbbb");
  strcpy(DizionarioTest[2], "bbbbbaa");
  strcpy(DizionarioTest[3], "abcdefg");
  strcpy(DizionarioTest[4], "z");

  printf("\nIl dizionario contiene le seguenti parole:");
  ElemDiz = 0;
  while(ElemDiz < NUM_PAROLE)
  {
    printf("\n%d: %s", ElemDiz, DizionarioTest[ElemDiz]);
    ++ElemDiz;
  }
  printf("\n");

  /*** eseguo qualche prova delle funzioni ***/
  ElemDiz = 0;
  while(ElemDiz < NUM_PAROLE)
  {
    printf("\ndistanza tra %s e %s = %d", DizionarioTest[0], DizionarioTest[ElemDiz], Distanza(DizionarioTest[0],
DizionarioTest[ElemDiz]));
    ++ElemDiz;
  }

  printf("\n\nEsiste(\"%s\") = %d", DizionarioTest[0], Esiste(DizionarioTest[0],
        DizionarioTest));
  /* nota: la notazione \" indica a printf di stampare a schermo il carattere
     'doppi apici' anzichÃ© considerarlo termine della stringa di controllo */
  printf("\nEsiste(\"no\") = %d", Esiste("nonono\0", DizionarioTest));
  printf("\nSuggerimento(\"%s\") = \"%s\"", DizionarioTest[0],
        DizionarioTest[Suggerimento(DizionarioTest[0], DizionarioTest)]);
  printf("\nSuggerimento(\"bbbbbba\") = \"%s\"",
        DizionarioTest[Suggerimento("bbbbbaa", DizionarioTest)]);
  printf("\nSuggerimento(\"bbbbbbz\") = \"%s\"",
        DizionarioTest[Suggerimento("bbbbbaz", DizionarioTest)]);
  printf("\nSuggerimento(\"abcde\") = \"%s\"",
        DizionarioTest[Suggerimento("abcde", DizionarioTest)]);
  printf("\n\n");

  return (0);
}
```

```
/* Nota: questo programma stampa

Il dizionario contiene le seguenti parole:
0: aaaaa
1: bbbbb
2: bbbbbaa
3: abcdefg
4: z

distanza tra aaaaa e aaaaa = 0
distanza tra aaaaa e bbbbb = 5
distanza tra aaaaa e bbbbbaa = 57
distanza tra aaaaa e abcdefg = 62
distanza tra aaaaa e z = 129

Esiste("aaaaa") = 1
Esiste("no") = 0
Suggerimento("aaaaa") = "aaaaa"
Suggerimento("bbbbbba") = "bbbbbaa"
Suggerimento("bbbbbbz") = "bbbbbaa"
Suggerimento("abcde") = "bbbbb"
*/

/* @@@@@@@@@@@@@@@@@@@@@@@@@@@ */


/*****************************
 Parte b.
*****************************/
int Esiste(Parola Par, Dizionario Diz)
{
  int ParolaInEsame;
  /* indice della parola di Diz attualmente in fase di confronto con Par */
  int ValoreUscita;
  /* valore da restituire al programma chiamante */

  ValoreUscita = 0;

  /*** confrontiamo Par con ciascuna delle parole di Diz */
  ParolaInEsame = 0;
  while ((0 == ValoreUscita) && (ParolaInEsame < NUM_PAROLE))
```

```
  {
    if (0 == Distanza(Par, Diz[ParolaInEsame]))
    {
      ValoreUscita = 1;
    }
    ++ParolaInEsame;
  }

  /* printf("\nDEBUG: FINITA RICERCA NEL DIZIONARIO DI %s", Par); */

  return (ValoreUscita);
}


/*****************************
 Parte c.
*****************************/
int Distanza(Parola Par1, Parola Par2)
{
  int Lettera;
  /* indice del carattere di Par1 e Par2 attualmente in fase di confronto */
  int DistanzaLettere;
  /* contiene la distanza tra i caratteri di Par1 e Par2 attualmente in fase di confronto */
  int DistanzaCalcolata;
  /* contiene la distanza tra Par1 e Par2 durante il suo calcolo */

  /* printf("\nDEBUG: STO PER CONFRONTARE LE PAROLE %s E %s", Par1, Par2); */

  DistanzaCalcolata = 0;
  Lettera = 0;

  while ((Par1[Lettera] != '\0') && (Par2[Lettera] != '\0'))
  /* fino a che non raggiungiamo la fine della prima tra le parole Par1 e Par2,
     o eventualmente di entrambe le parole se di lunghezza uguale */
  {
    /* printf("\nDEBUG: STO CONFRONTANDO %c con %c", Par1[Lettera], Par2[Lettera]); */

    DistanzaLettere = Par1[Lettera] - Par2[Lettera];
    /* dato che i due elementi da confrontare sono entrambi lettere minuscole
       e in C le lettere minuscole sono associate -nell'ordine- ad interi
       consecutivi, il confronto si riduce al calcolo di una differenza */
```

```
    /*** calcolo il valore assoluto ***/
    if (DistanzaLettere < 0)
    {
      DistanzaLettere = -1 * DistanzaLettere;
    }

    DistanzaCalcolata = DistanzaCalcolata + DistanzaLettere;
    ++Lettera;
  }

  if ('\0' != Par1[Lettera])
  /* se non abbiamo ancora raggiunto la fine di Par1: cio' accade
     quando Par2 e' piu' corta di Par1 */
  {
    while (Par1[Lettera] != '\0')
    /* fino a che non raggiungiamo anche la fine di Par1 */
    {
        /* printf("\nDEBUG: avanza '%c', aggiungo %d", Par1[Lettera], DIST_LETTERA_MANCANTE); */
        DistanzaCalcolata = DistanzaCalcolata + DIST_LETTERA_MANCANTE;
        ++Lettera;
    }
  }
  else
  /* in questo caso abbiamo raggiunto la fine di Par1, dunque (dal momento che
     abbiamo gia' raggiunto la fine di almeno una delle due parole) l'unica
     parola che potrebbe eventualmente non essere ancora terminata e' Par2 */
  {
    while (Par2[Lettera] != '\0')
    /* fino a che non raggiungiamo anche la fine di Par2; si noti che qualora
       anche par2 fosse gia' terminata questo ciclo -essendo a condizione
       iniziale- non viene eseguito nemmeno una volta */
    {
      /* printf("\nDEBUG: avanza '%c', aggiungo %d", Par2[Lettera], DIST_LETTERA_MANCANTE); */
      DistanzaCalcolata = DistanzaCalcolata + DIST_LETTERA_MANCANTE;
      ++Lettera;
    }
  }

  return (DistanzaCalcolata);
}
```

```
/*****************************
 Parte d.
*****************************/
int Suggerimento(Parola Par, Dizionario Diz)
/* NOTA: LA FUNZIONE Suggerimento RICHIEDE CHE Diz CONTENGA ALMENO UNA PAROLA */
{
  int ParolaInEsame;
  /* indice della parola di Diz attualmente in fase di confronto con Par */
  int ParolaMin;
  /* indice dell'elemento di Diz con la minima distanza da Par trovato finora */
  int DistanzaMin;
  /* minima distanza trovata finora tra Par e un elemento di Diz */
  int DistanzaAttuale;
  /* distanza tra Par e l'elemento di Diz attualmente in esame */

  /* inizialmente poniamo che la parola di Diz piÃ¹ simile a Par sia la prima;
     perche' quanto segue funzioni correttamente e' necessario che tale prima
     parola esista */
  ParolaMin = 0;
  DistanzaMin = Distanza(Par, Diz[0]);

  /*** confrontiamo Par con ciascuna delle parole successive alla prima */
  ParolaInEsame = 1;
  while (ParolaInEsame < NUM_PAROLE)
  {
    DistanzaAttuale = Distanza(Par, Diz[ParolaInEsame]);

    if (DistanzaAttuale < DistanzaMin)
    {
      ParolaMin = ParolaInEsame;
      DistanzaMin = DistanzaAttuale;
    }

    ++ParolaInEsame;
  }

  return (ParolaMin);
}
```