Univ. di Roma “La Sapienza” - Laurea in Informatica/Tecn. Inform.
Esame di Fondamenti di Programmazione/Programmazione 1
Andrea Sterbini – canale AD – 18-6-09
Compito A

Esercizio 1A (stringhe)

Secondo una ricerca recente, se in una frase si mescolano le lettere di ciascuna parola lasciando la prima e l'ultima lettera al loro posto, il cervello è capace ugualmente di ricostruire il significato delle parole e della frase. Si implementi una funzione di nome **`confondi`**, che riceve come argomento un testo (e tutti gli altri argomenti che ritenete necessari) e che ne produce una copia in cui, per ciascuna parola, le lettere interne sono mescolate (mantenendo la prima e l'ultima lettera della parola a posto). La funzione deve tornare la copia modificata del testo ed il numero di caratteri effettivamente spostati (cioè per i quali nella stessa posizione nella frase originale c'era un carattere diverso)

Esempio: la frase “la vecchia con la borsa salta il fosso senza rincorsa” potrebbe essere trasformata nella frase “la v**h**c**cie**a con la b**rso**a sa**tl**a il f**so**so s**zen**a r**rcoin**sa ” e la funzione deve tornare anche il valore 18 (i caratteri spostati, che ho indicato in grassetto nell'esempio).

NOTA: per generare numeri a caso si usi la funzione **`int random();`** che torna un numero intero casuale compreso tra 0 e MAX_INT inclusi.

Soluzione

L'esercizio si risolve prima copiando il testo originale, e poi cercando una per una le parole e scambiando a caso le lettere senza muovere la prima e ultima lettera. Una volta mescolate le lettere basta confrontare il nuovo testo con quello originale per contare quanti caratteri si sono effettivamente spostati. Per cercare le parole si può usare strtok oppure realizzare una funzione che cerca la posizione del prossimo separatore (spazio o '\0').

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
// mescolo le lettere interne alla parola
void mescola(char * testo, int start, int end) {
      int i, j;
      for (i=start+1 ; i<end-1 ; i++) {         // scandisco dal 2 al terzultimo
            j = (random() % (end - i)) + i;     // scelgo a caso un car. seguente
            char temp = testo[i];               // e li scambio
            testo[i] = testo[j];
            testo[j] = temp;
      }
}
// cerco la fine della parola a partire dalla posizione start
int prossimaParola(char * testo, int * start, int * end) {
      int found = 0;
      while ( testo[*start] == ' ')
            (*start) ++;
      for (*end = *start ; testo[*end+1] && testo[*end+1] != ' ' ; (*end) ++)
            found = 1;
      return found;
}
// cerco le parole e le mescolo
char * confondi(char * testo, int * nmossi) {
      int start = 0, end, i;
      int len = strlen(testo);
      char * copia = malloc(len+1);             // alloco spazio per la copia
      strcpy(copia, testo);                     // copio il testo
      while(prossimaParola(copia, &start, &end)) {    // cerco la prossima parola
            mescola(copia, start, end);         // la rimescolo
            start = end + 1;                    // aggiorno l'indice al seguito
      }
      *nmossi = 0;                              // conto i caratteri spostati
      for (i=0 ; i<len ; i++)
            if (testo[i] != copia[i])
                  (*nmossi) ++;
      return copia;
}
// programma principale
int main() {
      char * testo = "la vecchia con la borsa salta il fosso senza rincorsa";
      int nmossi;
      char * copia = confondi(testo, &nmossi);
      printf("testo: '%s'\ncopia: '%s'\nnmossi= %d\n", testo, copia, nmossi);
      return 0;
}
```

Esercizio 2A (liste e matrici)

Sia dato un tipo struct definito come segue:

```
typedef struct scopri {
     int x_1, y_1, x_2, y_2;
     struct scopri * succ;
} Scopri;
```

Vogliamo giocare a Memory. Si scriva una funzione **`trova_le_coppie`** che riceve in input una matrice di 10 righe e 4 colonne, in cui ci sono 2 coppie di ciascuno dei valori fra 1 e 10, e due liste di elementi di tipo **`Scopri`**, che rappresentano le mosse dei due giocatori. La funzione estrae nell'ordine una mossa da ciascuna lista (alternatamente) e, se i valori presenti nella matrice alle coordinate x_1 y_1 e x_2 y_2 si corrispondono, li cancella dalla matrice e assegna un punto al giocatore (che ha individuato una coppia). Quando tutte le coppie sono state trovate, oppure al termine di una delle liste, la funzione restituisce **1** se il primo giocatore ha più punti del secondo, **-1** se il secondo giocatore ha più punti del primo e **0** se hanno pareggiato.

Soluzione

Qui bisognava scorrere le liste a turno ed esaminare la matrice tenendo il conto delle coppie trovate ed aggiornando la matrice (ad esempio sostituendole con 0) per tener conto delle coppie trovate e tolte dal tavolo.

```
// controllo se ho trovato una coppia uguale
// torno 1 e aggiorno la matrice se sì
int controllaMossa ( Scopri * mossa, int M[10][4] ) {
      if (  M[mossa->y_1][mossa->x_1] == M[mossa->y_2][mossa->x_2]) {
            M[mossa->y_1][mossa->x_1] = 0;
            M[mossa->y_2][mossa->x_2] = 0;
            return 1;
      }
      else return 0;
}
int memory(int M[10][4], Scopri * mosse1, Scopri * mosse2) {
      Scopri * temp;
      int punteggio1 = punteggio2 = 0, coppie = 20, mossa;
      // se ci sono mosse da fare e coppie da trovare vado avanti
      while (mosse1 && mosse2 && coppie) {
            mossa = controllaMossa(mosse1, M);  // faccio la mossa del giocatore 1
            punteggio1 += mossa;                // ne conto il risultato
            coppie -= mossa;                    // e aggiorno il numero di coppie trovate
            temp = mosse1;                      // faccio il pop dalla lista 1
            mosse1 = mosse1->succ;
            free(temp);                         // e disalloco la mossa
            mossa = controllaMossa(mosse2, M);  // faccio la mossa del giocatore 2
            punteggio2 += mossa;                // ne conto il risultato
            coppie -= mossa;                    // e aggiorno il numero di coppie trovate
            temp = mosse2;                      // faccio il pop dalla lista 2
            mosse2 = mosse2->succ;
            free(temp);                         // e disalloco la mossa
      }
      return sign(punteggio1 - punteggio2);     // calcolo +1 o -1 o 0
}
```

Esercizio 3A (ricorsione)

Si spieghi brevemente (max una facciata) come si fa a trasformare un qualsiasi ciclo in una funzione ricorsiva che fa le stesse cose.

Si faccia un esempio concreto:

- prima implementando ITERATIVAMENTE la funzione **scarto_massimo** che calcola la differenza tra massimo e minimo di un vettore di interi,
- poi trasformando la funzione scarto_massimo nella corrispondente versione RICORSIVA

Soluzione

Ricordando che un ciclo è formato dagli elementi seguenti: inizializzazione, condizione, corpo, incremento e variabili di stato (ad esempio contatori)

Ogni ciclo può essere trasformato in una funzione ricorsiva con i passi seguenti:

- tutte le variabili di stato vengono inserite tra gli argomenti della funzione in modo da poterle aggiornare tra una chiamata ricorsiva e la successiva
- l'inizializzazione delle variabili di stato viene svolta dalla funzione chiamante, che passa gli opportuni valori iniziali negli argomenti della prima chiamata della funzione. Per questo conviene realizzare una funzione apposta che fa la sola prima chiamata della funzione ricorsiva con i valori giusti per le inizializzazioni.
- il test per verificare se il ciclo deve essere eseguito viene usato come test del caso base della funzione ricorsiva. Per questo il test deve essere rovesciato rispetto al ciclo originale.
- L'incremento o aggiornamento delle variabili di stato avviene passando alla successiva chiamata ricorsiva i valori opportunamente aggiornati degli argomenti.
- Le operazioni finali che normalmente sono eseguite dopo il ciclo vengono eseguite nel caso base

Esempio: calcolo dello scarto massimo iterativo e ricorsivo:

```
int scartomassimo( int V[], int dim) {
      int i, min = V[0] , max = V[0];     // inizializzo con un valore presente in V
      for (i = 1 ; i<dim ; i++) {               // per tutti gli altri elementi
            if (V[i] < min)   min = V[i];       // aggiorno il min se necessario
            if (V[i] > max)   max = V[i];       // aggiorno il max se necessario
      }
      return max - min;                         // alla fine torno lo scarto
}
// ricerca del min e max ricorsiva
int scartomassimo_ric( int V[], int dim, int i, int min, int max) {
      if (i >= dim)                 // test del ciclo a rovescio
            return max - min;       // parte finale dopo il ciclo
      else {            // il corpo del ciclo diventa la parte prima della ricorsione
            if (V[i] < min)         min = V[i];
            if (V[i] > max)         max = V[i];
            return scartomassimo_ric( V, dim, i+1, min, max);
      }
}
// funzione di utilità che inizializza i, min e max con i valori giusti
int scartomassimo_R( int V[], int dim) {
      // parto dal secondo elemento ed inizializzo min e max col primo elemento
      return scartomassimo_ric( V, dim, 1, V[0], V[0]);
}
int main () {
      int V[] = {3, 6, 11, 23, 9 , 1, 74}, dim = 7;
      printf("scarto = %d (iter) %d (ric)", scartomassimo(V,dim),scartomassimo_R(V,dim));
}
```

Esercizio 4A (alberi binari di ricerca)

Sia dato un tipo struct definito come segue:

```
typedef struct nodo {
      long val;
      struct nodo * figlio_sx;
      struct nodo * figlio_dx;
} Nodo;
```

Scrivere una funzione **sequenza_massima** che riceve come argomenti: un albero binario di ricerca per mezzo di un puntatore a Nodo (e tutti gli altri argomenti che ritenete necessari) e restituisce il massimo numero di valori che formano una sequenza crescente nell'albero (per sequenza crescente si intende che i valori [x, x+1, x+2, ..., x+k] compaiono nell'albero; la lunghezza di questa sequenza è k+1).

Esempio: Se nell'albero sono presenti i valori 1, 7, 12, 13, 15, 16, 17, la funzione deve restituire il valore 3 (visto che nell'albero è presente la sequenza di tre elementi consecutivi 15, 16, 17).

Soluzione

Per cercare la sequenza massima bisogna ricordarsi che un albero binario di ricerca, se visitato **inordine,** produce la sequenza ordinata dei valori contenuti in esso.

Quindi, se passiamo opportune variabili ad una funzione ricorsiva che fa la visita inordine, possiamo individuare la sequenza più lunga. Conviene passare queste variabili per riferimento, in modo che vengano aggiornate dalla funzione. Per riconoscere la sequenza più lunga sono sufficienti tre variabili che contengono: la massima sequenza trovata fino ad un certo punto, l'ultimo numero letto e la lunghezza della sequenza corrente. Per ogni nuovo numero dell'albero possiamo capire se la sequenza corrente è terminata o se continua confrontandolo con l'ultimo valore letto precedentemente. Inoltre, ogni volta che una sequenza si allunga è possibile aggiornare il massimo se necessario.

```
void sequenzamassima(Nodo * albero, int * last, int * max, int * count) {
      // se esiste almeno un elemento nell'albero
      if (albero) {
            // visito il sottoalbero sinistro
            sequenzamassima(albero->figlio_sx, last, max, count);
            // poi esamino il nodo radice
            if (albero->val - 1 == (*last)) {   // se la sequenza continua
                  (*count) ++;                  // incremento la lunghezza
                  if (*count > *max)
                        *max = *count;          // aggiorno il max se serve
            } else {                            // altrimenti ne inizia una nuova
                  *count = 1;                   // con solo questo elemento
            }
            // poi proseguo la visita sul figlio destro
            sequenzamassima(albero->figlio_dx, albero->val, max, count);
      }
      // se l'albero è vuoto non faccio nulla
      // (così gestisco automaticamente anche sottoalberi SX e DX vuoti)
}
// funzione di utilità che inizializza count, max e last
int sequenzamassima_util(Nodo * albero) {
      if (albero) {
            int count = 0;
            int max = 0;
            // last può essere inizializzato con il valore della radice
            int last = albero->val;
            sequenzamassima(albero, &last, &max, &count);
            return max;
      } else
            return 0;
}
```

Univ. di Roma “La Sapienza” - Laurea in Informatica/Tecn. Inform.
Esame di Fondamenti di Programmazione/Programmazione 1
Andrea Sterbini – canale AD – 18-6-09
Compito B

Esercizio 1B (stringhe)

Si deve cercare una sottostringa in un testo. Si supponga che la stringa da cercare nel testo possa contenere dei caratteri errati e che i caratteri possono essere errati SOLO sul loro bit meno significativo. Si implementi la funzione **`trova`** che cerca la stringa nel testo ignorando il valore del bit meno significativo (quindi due caratteri sono uguali se al più differiscono per il LSB bit)

La funzione deve tornare, tramite due variabili **passate per riferimento**:

- la posizione di inizio della prima corrispondenza trovata della sottostringa nel testo
- il numero di caratteri sbagliati (quelli che corrispondono ma hanno il bit meno significativo diverso)

Soluzione

Questo è il solito esercizio in cui si cerca una stringa in un testo con la sola differenza che due caratteri sono uguali se lo sono i loro 7 bit più significativi. Per confrontare questi 7 bit si possono usare diversi modi, ad esempio dividere il valore per 2 (divisione intera), oppure fare l'AND bit a bit con il valore binario 11111110 (ovvero 254 decimale o 0xFE esadecimale o 0376 ottale).

```
// torno la posizione o -1 e inoltre il numero di caratteri diversi (per il LSB)
int cerca(char * testo, char * stringa, int * n_sbagli) {
     int i, j;
     for (i=0 ; testo[i] ; i++) {          // per ogni possibile posizione i nel testo
          * n_sbagli = 0;                              // comincio a contare gli sbagli
          for (j=0 ; stringa[j] && testo[i+j] ; j++) // per ogni carattere j della stringa
               if (testo[i+j]/2 != stringa[j]/2)     // se il carattere corrente è diverso
                    break;                           // smetto e vado avanti nel testo
               else                                  // altrimenti avanzo nella stringa
                    if (testo[i+j] != stringa[j])    // e conto il carattere se è diverso
                         (*n_sbagli)++;
          // sono qui se il testo è finito o se è finita la stringa
          if (stringa[j] == '\0')                   // se la stringa è finita
               return i;                            // torno la posizione corrente
     }
     return -1;                            // se sono qui le ho provate tutte invano
}
```

Esercizio 2B (liste e matrici)

Sia dato un tipo struct definito come segue:

```
typedef struct coordinata {
      int x;
      int y;
      struct coordinata * succ;
} Coordinata;
```

Scrivere una funzione **battaglia_navale** che riceve in input due matrici di interi di dimensioni 10 per 10, contenenti valori 0 o 1, che rappresentano la posizione delle navi (0 acqua, 1 nave) e due liste di coordinate, dove ogni coordinata rappresenta il punto in cui si vuole sparare. La somma degli 1 in ogni matrice è pari a 25 (1 nave da 5, 1 da 4, 2 da 3, 3 da 2 e 4 da 1).

La funzione estrae nell'ordine i valori dalle liste (in maniera alternata) e restituisce 1 se il primo giocatore distrugge per primo tutte le navi dell'avversario, -1 se il secondo giocatore distrugge per primo tutte le navi dell'avversario, restituisce 0 se arriva al termine di una delle due liste senza che alcun giocatore abbia battuto l'avversario.

Soluzione

Questo è talmente simile all'altro che vi invito a guardare quello.

Esercizio 3B (sort e ricorsione)

Si implementi RICORSIVAMENTE la funzione **merge**, che fonde due vettori ordinati di interi di uguale lunghezza copiandone i contenuti in un terzo vettore (di dimensioni doppie) ordinato.

Si implementi RICORSIVAMENTE l'algoritmo di **mergesort** di un vettore di interi usando la funzione merge e tutte le altre strutture dati o argomenti che ritenete necessari.

Soluzione

Questo esercizio l'abbiamo fatto in classe sia con i vettori che con le liste.

```
// fondo i vettori ordinati A e B copiandone gli elementi in C
// uso gli argomenti dimA, dimB e dimC per indicare quanti elementi vanno ancora spostati
void merge(int A[], int B[], int C[], int dimA, int dimB, int dimC) {
      if (!dimC)  return;           // se non ci sono più elementi da copiare esco
      // se non ce ne sono in B li copio da A
      if (!dimB) {
            C[0] = A[0];            // copio da A a C e ricorro aggiornando A C dimA dimC
            return merge(A+1, B, C+1, dimA-1, dimB, dimC-1);
      }
      // se non ce ne sono in A li copio da B
      if (!dimA) {
            C[0] = B[0];            // copio da B a C e ricorro aggiornando B C dimB dimC
            return merge(A, B+1, C+1, dimA, dimB-1, dimC-1);
      }
      // se sono arrivato qui ci sono elementi sia in A che in B, confronto il primo
      if (A[0]<B[0]) {
            C[0] = A[0];            // copio da A a C e ricorro aggiornando A C dimA dimC
            return merge(A+1, B, C+1, dimA-1, dimB, dimC-1);
      } else {
            C[0] = B[0];            // copio da B a C e ricorro aggiornando B C dimB dimC
            return merge(A, B+1, C+1, dimA, dimB-1, dimC-1);
      }
}
```

NOTA: compattando un po' i test è possibile ridurre i casi a 3

Mergesort ha bisogno di un vettore di appoggio per eseguire il passo merge dei mezzi vettori ordinati per metterli nel vettore destinazione, è sufficiente un unico vettore delle stesse dimensioni di quello disordinato e di quello in cui vanno messi i risultati ordinati perchè ciascuna chiamata ricorsiva lavora su un segmento separato dei vettori.

```
void mergesort(int disordinati[], int ordinati[], int appoggio[], int dim) {
      // casi base: se il vettore da ordinare ha zero o un solo elemento è già ordinato
      if (dim == 0)
            return;                              // non fo nulla
      if (dim == 1)
            ordinati[0] = disordinati[0];        // copio il valore nella destinazione
      else {
            // altrimenti devo spezzare il vettore in 2
            int meta1 = dim/2;
            // per via della divisione intera la seconda parte potrebbe essere diversa
            int meta2 = dim – meta1;
            // ordino la prima metà mettendo il risultato in appoggio
            mergesort(disordinati, appoggio, ordinati, meta1);
            // ordino la seconda metà mettendo il risultato in appoggio (seconda metà)
            mergesort(disordinati+meta1, appoggio+meta1, ordinati+meta1, meta2);
            // fondo le due metà ordinate mettendo i risultati nel vettore finale
            merge(appoggio, appoggio+meta1, ordinati, meta1, meta2, dim);
      }
}
// esempio di main
int main () {
      int dim = 9, i;
      int disordinati[] = {9, 4, 2, 44, 92, 3, 17, -3, 12};
      int ordinati[dim];
      int appoggio[dim];
      mergesort(disordinati, ordinati, appoggio, dim);
      for (i=0 ; i<dim ; i++)
            printf(“%d ”, ordinati[i]);
      printf(“\n”);
}
```

Esercizio 4B (alberi binari di ricerca)

Sia dato un tipo struct definito come segue:

```
typedef struct nodo {
      long val;
      struct nodo * figlio_sx;
      struct nodo * figlio_dx;
} Nodo;
```

Si scriva la funzione **somma_mancanti** che riceve in input: un albero binario di ricerca con elementi di tipo Nodo e restituisce la somma dei valori interi mancanti all'interno dell'albero, compresi fra il valore massimo e il valore minimo.

Esempio: Se nell'albero sono presenti i valori 1, 7, 12, 13, 15, 16, 17, la funzione deve restituire il valore 72 (che è la somma dei valori 2, 3, 4, 5, 6, 8, 9, 10, 11, 14 mancanti).

Soluzione

Qui possiamo calcolare la somma dei mancanti sottraendo la somma dei numeri presenti nell'albero dalla somma di tutti i numeri compresi tra il minimo ed il massimo. Quindi possiamo calcolare, con una sola visita di qualsiasi tipo sull'albero minimo, massimo e somma dei presenti, e poi calcolare il risultato.

(in realtà il minimo e il massimo si possono trovare più rapidamente, ma visto che devo sommare tutti i nodi...)

Passo i tre argomenti per riferimento per poterli aggiornare in ciascuna chiamata ricorsiva.

Per calcolare la somma di tutti i numeri interi tra min e max inclusi posso usare un ciclo oppure la formula

$$(max-min+1)*(min+max)/2$$

```
void somma_min_max(Nodo * albero, int * min, int * max, int *somma) {
      if (albero) {                                       // se c'è almeno un nodo
            if (albero->val > *max) *max = albero->val;     // aggiorno il max
            if (albero->val < *min) *min = albero->val;     // aggiorno il min
            *somma += albero->val;                          // aggiorno la somma
            somma_min_max(albero->figlio_sx, min, max, somma);    // visito il SX
            somma_min_max(albero->figlio_dx, min, max, somma);    // visito il DX
      }
      // se l'albero è nullo non faccio niente
      // questo mi evita di dover controllare se i figli sono NULL
}
// calcolo la somma dei numeri mancanti tra min e max
int somma_mancanti(Nodo * albero) {
      if (!albero)                                    // se l'albero è vuoto ...
            return 0;
      int min = albero->val;        // inizializzo min e max con un valore dell'albero
      int max = albero->val;
      int somma = 0;
      somma_min_max(albero, &min, &max, &somma);
      // la somma dei numeri da min a max compresi è        (max-min+1)*(min+max)/2
      return (max-min+1)*(min+max)/2 - somma;
}
```