Esercitazione 2: stringhe e liste

Irene Finocchi
finocchi@di.uniroma1.it

Esercizio 1

Data una stringa che rappresenta un intero, restituire il valore corrispondente

```
int my_atoi(char *s) {
  int val=0;
  while (*s != '\0') {
     val = (*s-'0') + val*10;
     s++;
  }
  return val;
}
```

® Esercizio 1

Data una stringa che rappresenta un intero, restituire il valore corrispondente

```
int myRec_atoi(char *s) {
  int len = strlen(s);
  if (len == 0) return 0;
  return myRec_atoi(s+1)+
         (*s-'0')*(int)pow(10,len-1);
}
```

Efficienza?

® Esercizio 1

Soluzione standard

Calcolare la lunghezza esternamente

```
int my_atoi(char *s) {
  int len = strlen(s);
  return myRec_atoi(s,len);
}
int myRec_atoi(char *s, int len) {
  if (len == 0) return 0;
  return myRec_atoi(s+1,len-1)+
       (*s-'0')*(int)pow(10,len-1);
}
```

Due passate sulla stringa

® Esercizio 1

Soluzione che effettua una sola passata

```
int my_atoi(char *s) {
  int val=0;
  myRec_atoi(s,&val);
  return val;
}
void myRec_atoi(char *s, int *val) {
  if (*s == '\0') return;
  *val = (*val)*10 + (*s-'0');
  myRec_atoi(s+1,val);
}
```

Esercizio 2

Verificare se una stringa è quasi-palindroma (al più un carattere "errato")

```
int quasi_palindroma(char *s) {
  int i, j = strlen(s)-1;
  for (i=0; s[i]==s[j] && i<=j; i++, j--);
  if (i>j) return 1;
  i++; j--;
  for ( ; s[i]==s[j] && i<=j; i++, j--);
  return ((i>j)?1:0);
}
```

® Esercizio 2

Verificare se una stringa è quasi-palindroma (al più un carattere "errato")

```
int quasi_palindroma(char *s, int len) {
  if (len<=1) return 1;
  if (s[0] == s[len-1])
    return quasi_palindroma(s+1, len-2);
  else return palindroma(s+1, len-2);
}
```

Corretto, ma non molto elegante

® Esercizio 2

Soluzione elegante

Risolviamo un problema più generale (al più k caratteri "errati")

```
int quasi_palindroma(char *s, int len, int k) {
  if (k<0) return 0;
  if (len<=1) return 1;
  if (s[0] == s[len-1])
      return quasi_palindroma(s+1,len-2,k);
  return quasi_palindroma(s+1, len-2, k-1);
}
```

Tipi per la gestione di liste

```
struct nodolista{
  int elem;
  struct nodolista *next;
};

typedef struct nodolista *ListPtr;
```

Esercizio 3 Data una lista L, rimuovere gli elementi duplicati

Sia x il primo elemento di L

1) Rimuovi tutte le occorrenze di x in L->next
   Sia L' la lista risultante

2) Rimuovi tutti i duplicati da L'
   Sia L'' la lista risultante

Restituisci la concatenazione di x e L''

Ⓡ Esercizio 3 Rimozione dei duplicati

Procedura ausiliaria per rimuovere le occorrenze di val

```
ListPtr rimuoviOcc(ListPtr L, int val) {
  if (!L) return NULL;
  if (L->elem == val) {
     ListPtr temp = L;
     L = rimuoviOcc(L->next, val);
     free(temp);
  }
  else L->next = rimuoviOcc(L->next, val);
  return L;
}
```

Ⓡ Esercizio 3 Rimozione dei duplicati

```
ListPtr rimuoviDup(ListPtr L) {
  if (!L) return NULL;

  L->next = rimuoviOcc(L->next, L->elem);
  L->next = rimuoviDup(L->next);
  return L;
}
```

Possiamo evitare di usare un metodo ausiliario?

Esercizio 3 — Un solo metodo ricorsivo

```
ListPtr rimuoviDup(ListPtr L, int flag, int val) {
  if (!L) return NULL;
  if (flag==1) {
      L->next = rimuoviDup(L->next, 0, L->elem);
      if (L->next!=NULL)
            L->next = rimuoviDup(L->next, 1, -1);
  }
  else if (L->elem == val) {
      ListPtr temp = L;
      L = rimuoviDup(L->next, 0, val);
      free(temp);

  } else L->next = rimuoviDup(L->next, 0, val);
  return L;
}
```

Esercizio 3

```
Input:   1 2 3 1 2 3
flag = 1   val = -1   1 2 3 1 2 3
  flag = 0   val = 1   2 3 1 2 3
    flag = 0   val = 1   3 1 2 3
      flag = 0   val = 1   1 2 3
        flag = 0   val = 1   2 3
          flag = 0   val = 1   3
  flag = 1   val = -1   2 3 2 3
    flag = 0   val = 2   3 2 3
      flag = 0   val = 2   2 3
        flag = 0   val = 2   3
    flag = 1   val = -1   3 3
      flag = 0   val = 3   3
Output: 1 2 3
```

Esempio di esecuzione

Esercizio 3 — Calcolo della lista in un parametro

```
void rimuoviDup(ListPtr *L, int flag, int val) {
  if (!(*L)) return NULL;
  if (flag==1) {
    rimuoviDup(&((*L)->next), 0, (*L)->elem);
    if ((*L)->next!=NULL)
    rimuoviDup(&((*L)->next), 1, -1);
  }
  else if ((*L)->elem == val) {
    ListPtr temp = *L;
    *L = (*L)->next;
    free(temp);
    rimuoviDup(L, 0, val);
  } else rimuoviDup(&((*L)->next), 0, val);
}
```

Homework 1

1) Data una stringa s, calcolare una nuova stringa in cui tutte le lettere ‘a’ accentate sono rimpiazzate dalla ‘a’ senza accento

2) Verificare se una stringa s1 e' sottostringa di s2

® Homework 1

3) Dati una lista L ed un intero k, rimuovere da L un elemento ogni k (ovvero, rimuovere gli elementi in posizione k, 2k, 3k, etc). La lista L modificata va calcolata in un parametro. E' ammesso fare deallocazioni, ma non allocazioni di nuova memoria.

Il metodo deve usare una sottoprocedura ricorsiva con il seguente prototipo:

```
void EliminaOgniKRec(ListPtr *L, int h, int k);
```

La sottoprocedura elimina un elemento ogni k, esclusi i primi h elementi della lista.