Programmazione

Giovanni Da San Martino

Dipartimento of Matematica, Università degli Studi di Padova
giovanni.dasanmartino@unipd.it
A.A. 2021-2022

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Previously on Programmazione

- In una prova per induzione:
- si dimostra la POST per il caso base
- assumendo vera la POST per le chiamate ricorsive all'interno della funzione, si dimostra la POST per il caso ricorsivo

$$\frac{P(0) \qquad P(n) \Rightarrow P(n+1))}{\forall n \,.\, P(n)}$$

```
int lunghezza_stringa(char *p){
   /* PRE: p è un puntatore a stringa
        POST: restituisce L(p) la lunghezza della stringa puntata da p, ovvero il numero di caratteri escluso \0  */
    if(*p == '\0')
        return 0; //CASO BASE: L(p)=0
    else { int l = lunghezza_stringa(p+1); // L(p+1)=l
        return l + 1; //la stringa puntata da p è composta da un carattere iniziale più la stringa puntata da p+1, per cui L(p) = L(p+1) + 1=l+1, quindi la POST è verificata
    }
}
```

Trasformazione da Ricorsione a Iterazione


```
tipo F_ric(tipo x) {

   if (casobase(x)) {
         istruzioni_casobase;
         return risultato;
   } else {
         istruzioni_nonbase;
         return F_ric(riduciComplessita(x));
   }
}
```

```
tipo F_iter(tipo x) {
while (!casobase(x)) {
         istruzioni_non_base;
         x = riduci_complessità(x);
}
istruzioni_casobase;
return caso_base;
```

Esempio di trasformazione


```
int confronta_array(int *X, int *Y, int dim) {
  if (dim==0)
          return 1;
  else {
    if (X[0]!=Y[0])
       return 0;
    else {
       return confronta_array(X+1, Y+1, dim-1);
    }
  }
}
```

```
int confronta_array(int *X, int *Y, int dim) {


while (dim!=0) {
   if (X[i]!=Y[i])
       return 0;
}
  return 1;
}
```

Esempio di Dimostrazione per Induzione


```
int potenza(int base, int esp) {
    /*
        PRE: esp>=0, base!=0
        POST: restituisce base^esp
    */
    if (esp==0) {
        return 1;
    } else {
        return base*potenza(base, esp-1);
    }
}
```

Caso base: ?

Caso ricorsivo: ?

Esempio di Dimostrazione per Induzione


```
int potenza(int base, int esp) {
    /*
        PRE: esp>=0, base!=0
        POST: restituisce base^esp
    */
    if (esp==0) {
        return 1;
    } else {
        return base*potenza(base, esp-1);
    }
}
```

Caso base: 1=base^0

Caso ricorsivo: ?

Esempio di Dimostrazione per Induzione


```
int potenza(int base, int esp) {
    /*
        PRE: esp>=0, base!=0
        POST: restituisce base^esp
    */
    if (esp==0) {
        return 1;
    } else {
        return base*potenza(base, esp-1);
    }
}
```

Caso base: 1=base^0

Caso ricorsivo: se esp>0
potenza(base, esp-1) restituisce base^esp-1 (questa prende il nome di ipotesi induttiva),

quindi

base*base^esp-1 = base*esp => POST

Esercizi

Progettare ed implementare una delle seguenti funzioni ricorsive (a seconda dove siete) e dimostrate la sua correttezza

1. int trova_elemento(int X[], int dim, int elem) (SIETE A CASA)
   /* POST restituisce 1 se elem è in X; 0 altrimenti*/

2. int array_pari(int X[], int dim) (SEDETE ALLA MIA SINISTRA)
   /* POST restituisce 1 se tutti gli elementi di X sono pari; 0 altrimenti */

3. int conta_occorrenze(int X[], int dim, int x) (SEDETE ALLA MIA DESTRA)
   /* POST restituisce il numero di occorrenze di x in X */

File Multipli

- Quando i nostri programmi diventano di grandi dimensioni, o per riutilizzare agevolmente funzioni già realizzate, è possibile implementare queste ultime in un file separato, che poi andremo a “collegare” al nostro file principale.
- In realtà si creano due file file
  - .c con l’implementazione delle funzioni
  - .h con le intestazioni (i prototipi) delle funzioni
- Abbiamo già visto alcuni esempi: stdio.h, assert.h
- Nel programma principale, per poter utilizzare le funzioni aggiuntive, basta utilizzare la direttiva #include
  - #incluide <stdio.h> (<> fanno si che si cerca tra i file forniti dal sistema operativo)
  - #include “stringhe.h” (“” cerca stringhe.h prima nella cartella corrente)

File Multipli

- Se il file header.h contiene
  - char *test (void);
- E nel nostro file principale

```
#include “header.h”
int main (void) {
printf(“%s\n”, test);
}
```

- Allora il preprocessore trasformerà il file principale in

```
char *test (void);
int main (void) {
   printf(“%s\n”, test);
}
```

File Multipli

- Allora il preprocessore trasformerà il file principale in

```
char *test (void);
int main (void) {
    printf("%s\n", test);
}
```

- Notate che non abbiamo ancora a disposizione l'implementazione di test. Questo è accettabile nella fase di compilazione, ma l'implementazione della funzione test deve essere raggiungibile nella fase finale di linking
- Questo permette la compilazione separata (ma non l'esecuzione!) dei vari file che costituiscono un progetto

Evitare Inclusione Multipla

- Includere un file header più volte corrisponde ad un errore di compilazione (si dichiara due volte la stessa funzione)
- Il C mette a disposizione delle direttive del preprocessore per evitarlo

```
#ifndef HEADER_FILE
#define HEADER_FILE

//contenuto del file header.h: prototipi di funzioni e #define

#endif
```

- le istruzioni tra ifndef ed endif vengono copiate solo se la variabile HEADER_FILE non è definita

In pratica

- Se si vuole creare un file con una serie di funzioni, per esempio che operano su stringhe, si deve
- creare il file .h con i prototipi delle funzioni e le #define (proteggere il file dall'inclusione multipla)
- creare un file .c (con lo stesso nome) che includa il file .h (così il compilatore controlla che le dichiarazioni di funzione sono coerenti)
- Nella fase di compilazione è sufficiente elencare i file .c che vogliamo compilare (non è necessario includere i .h)

gcc –o programma main.c stringhe.c

- Se i file da compilare sono molti, conviene utilizzare l'utility make per sveltire il processo di compilazione (richiede un file di configurazione, Makefile, nel quale in pratica si specifica il comando gcc)

Esercitazione 1

- Correttezza: prestate attenzione ai casi particolari (accesso ad elementi di un vettore oltre la loro dimensione) oppure l'utilizzo di una chiave negativa
- Efficienza: a parte dettagli non erano possibili algoritmi significativamente più veloci di altri
- Organizzazione del codice: poiché la chiave di cifratura può essere negativa, è conveniente utilizzare una sola funzione per la codifica e la decodifica (è sufficiente invocare la funzione di codifica con -1*k per decodificare)
- Stile: I commenti (e le PRE/POST) hanno lo scopo minimizzare il tempo per capire il vostro codice. Se mettete troppi commenti, si impiegherà più tempo per leggere il vostro codice

Esercizi

- Cosa stampa il seguente codice?

```
int x;
int y=2;
int *p, *q = &y;
int **qq = &p;
**qq = 3;
printf("%d\n", **qq);
```