ALGORITMI E STRUTTURE DATI

Esercitazioni

Andrea Orlandini
http://www.dia.uniroma3.it/~orlandin/asd/
e-mail: orlandin@dia.uniroma3.it

Orario di ricevimento: Martedì 14.00 - 16.00

Introduzione al C
+
Array

StudentiDIA Forum

http://forum.studentidia.org/

Dove trovare i lucidi presentati a lezione

http://limongelli.dia.uniroma3.it/asd/

Obiettivi delle esercitazioni

Dato un problema, utilizzare le metodologie studiate a lezione per:

- Scegliere le strutture adatte per rappresentare i dati
  - Quale si adatta meglio alla natura del problema
- Progettare un algoritmo che risolva il problema
  - Scrivere un algoritmo sfruttando le funzioni legate alle strutture scelte
- Produrre un programma (funzionante) in linguaggio C
  - Tradurre le azioni dell'algoritmo in istruzioni del linguaggio C

Caratteristiche del Linguaggio C

- linguaggio C [Ritchie] sviluppato negli anni '70 in ambiente UNIX
  - Medio livello
  - Compilato
  - Imperativo
  - Per programmatori

- utilizzato per programmazione di sistema (sistemi operativi,compilatori, etc)

- ANSI C [Kernighan,Ritchie] linguaggio standard

Struttura generale di un programma C

```
include delle librerie

dichiarazioni globali

tipo_di_ritorno main (elenco_di_argomenti)
{
        sequenza di istruzioni
}

tipo_di_ritorno f1 (elenco_di_argomenti)
{
        sequenza di istruzioni
}

tipo_di_ritorno f2 (elenco_di_argomenti)
{
        sequenza di istruzioni
}

tipo_di_ritorno f3 (elenco_di_argomenti)
{
        sequenza di istruzioni
}
```

Identificatori e Tipi di Base

Identificatori

- No parole riservate (*if*, *else*, *float*, *double*, *while*, *do*,...)
- Primo carattere deve essere una lettera o un carattere di sottolineatura (_)
- I restanti possono essere caratteri, numeri o caratteri di sottolineatura

P.es. *conta*, *ver234* e *altro_id* sono identificatori validi mentre *1conta*, *un!identificatore* e *altro..identificatore* non lo sono

Tipi di base

- *char* [8 bit] per caratteri ASCII.
- *int* [16 bit] per numeri che non richiedono una parte frazionaria.
- *float* [32 bit] e double [64 bit] per memorizzare numeri reali.
- *void* corrisponde ad un tipo indefinito.

il C permette di definire nuovi nomi per i tipi tramite l'istruzione *typedef*:

typedef tipo nome;

P. es. il tipo bool non esiste. Definiamolo tramite il tipo int

```
typedef int bool;
```

Variabili e Funzioni

Variabili

locazione di memoria a cui è assegnato un nome
dichiarazione di una variabile:

tipo elenco_di_identificatori;

per esempio

```
int i,j,k;

double bilancio, profitto=0.0, perdita=0.0;

typedef bool int;
#DEFINE TRUE  1
#DEFINE FALSE 0
bool variabile_booleana;
```

Le variabili possono essere dichiarate all'interno delle funzioni, nella definizione degli argomenti di una funzione (LOCALI) e all'esterno di tutte le funzioni (GLOBALI).

Funzioni

Le funzioni hanno un identificatore, un tipo di ritorno e una lista di parametri. Il tipo di ritorno è il tipo del valore restituito dalla funzione, la lista di parametri è una serie di definizioni di variabili.

Attenzione, il passaggio delle variabili come parametri è fatto SEMPRE per valore.

Esempi di Funzioni

```
void funzione1(int n)
{
      int i;
      for(i=0;i<n;i++) printf(‘‘%d’’,i);
}
```

```
int funzione1()
{
      int i,sum=0;
      for(i=0;i<10;i++) sum=sum+i;
      return sum;
}
```

```
int i, n=10;

void main()
{
      for(i=0;i<n;i++) printf(‘‘%d’’,i);
}
```

Strutture di controllo

- ```
  if (condizione) istruzione;
                  [else istruzione;]
  ```
- ```
  switch (variabile)
  {
        case cost1: sequenza_di_istruzioni;[break;]
        case cost2: sequenza_di_istruzioni;[break;]
        ...
        case costN: sequenza_di_istruzioni;[break;]
        default:    sequenza_di_istruzioni;
  }
  ```
- ```
  for(inizializzazione; condizione; incremento)
         sequenza_di_istruzioni;
  ```
- ```
  while (condizione)
         sequenza_di_istruzioni;;
  ```
- ```
  do sequenza_di_istruzioni;
  while (condizione);
  ```

Operatori relazionali: $>$, $>=$, $<$, $<=$, $==$, $!=$

Operatori logici: &&, ||, !
  ```

tipo nome_var[dimensione];

```
int conta[10];
```

Un array di caratteri compone una stringa (Carattere finale '$\backslash 0$'):

```
char str[10] = ''ALGORITMI'';
```

Array a due o più dimensioni sono definiti in maniera analoga:

tipo nome_var[dim1][dim2]...[dimN];

Note all'uso degli array:

- Tramite indice si accede ai valori contenuti nel vettore e si possono modificare

```
char A[10];
A[1]='H';
printf(``%c'',A[1]);
```

- Useremo nel seguito la seguente definizione di Vettori:

```
#define NumElementi 100
typedef ... TipoElemVettore; /* p.es. int,char,etc */
typedef TipoElemVettore TipoVettore[NumElementi];
```

- Il passaggio di un array come parametro di una funzione avviene sempre per riferimento
- E' possibile inizializzare un vettore (ipotesi TipoElemVettore = int)

```
TipoElemVettore vet[5] = {1,2,3,4,5};
```

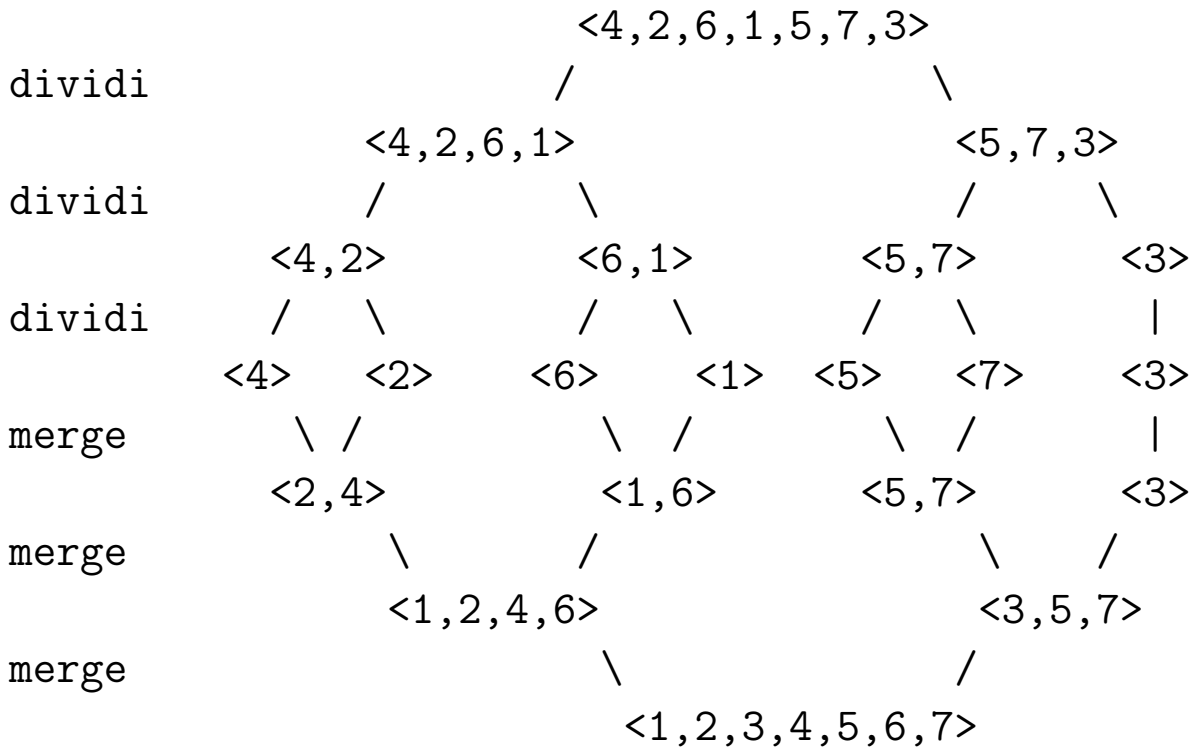
Merge sort - Ordinamento per fusione

Per ordinare la sequenza A di n elementi:

Divide: dividere A in due sottosequenze A_1 e A_2 di $n/2$ elementi

Impera: ordinare, ricorsivamente,A_1 e A_2

Combina: *fondere* le due sequenze ordinate ottenute



Merge Sort (II)

Fusione: sotto le ipotesi che:

A è un array

$p \leq q < r$ sono indici dell'array A

$A[p..q]$ e $A[q+1..r]$ sono ordinati

MERGE(A,p,q,r) fonde $A[p..q]$ e $A[q+1..r]$, generando $A[p..r]$ ordinato

MERGE-SORT(A,p,r): ordina gli elementi di $A[p..r]$

Richiamato inizialmente con MERGE-SORT(A,1,$length[A]$).

```
MERGE-SORT(A,p,r)
1.  if p < r
2.     then q <- (p+r) div 2
3.          MERGE-SORT(A,p,q)
4.          MERGE-SORT(A,q+1,r)
5.          MERGE(A,p,q,r)
```

1. Se $p \geq r$, $A[p..r]$ è vuoto oppure ha un solo elemento: il sottoarray è già ordinato.
2. `(p+r) div 2` indica la divisione intera $= \lfloor (p+q)/2 \rfloor$.

3.-4. Ordinamento ricorsivo dei due sottoarray.

5. Fusione.

Fusione

Ipotesi: $p \leq q < r$ sono indici dell'array A; $A[p..q]$ e $A[q+1..r]$ sono ordinati

```
MERGE(A,p,q,r)
i <- p      j <- q+1          k <- 1
while i<=q e j<=r
     do if A[i]<=A[j] then APP[k] <- A[i]
                           i <- i+1
                      else APP[k] <- A[j]
                           j <- j+1
        k <- k+1
if i=q+1 then for i <- j to r
                 do APP[k] <- A[i]
                    k <- k+1
         else for j <- i to q
                 do APP[k] <- A[j]
                    k<-k+1
k <- 1
for i <- p to r
   do A[i] <- APP[k]
      k <- k+1
```

MergeSort.c per MergeRicorsivo

MergeVettore.c per fondere due sottovettori

```
void MergeRicorsivo(TipoVettore A, int iniziale, int finale)
  /* Ordina gli elementi del vettore A di indice compreso tra iniziale e
     finale usando l'algoritmo di ordinamento per fusione. */
{
  int mediano;

  if (iniziale < finale)    /* l'intervallo da iniziale a finale, estremi inclusi
  {                            comprende almeno due elementi */
    mediano = (iniziale + finale) / 2;

    MergeRicorsivo(A, iniziale, mediano);
    MergeRicorsivo(A, mediano+1, finale);

    MergeVettore(A, iniziale, mediano, finale);
  }
}  /* MergeRicorsivo */

void MergeSort(TipoVettore A, int n)
  /* Ordina i primi n elementi del vettore A usando l'algoritmo
     di ordinamento per fusione. */
{
  MergeRicorsivo(A, 0, n-1);
}  /* MergeSort */
```

```
void MergeVettore(TipoVettore A, int iniziale, int mediano, int finale)
  /* Fonde i due sottovettori ordinati di A da iniziale a mediano e
     da mediano+1 a finale in un unico sottovettore ordinato. */
{
  TipoVettore B;     /* vettore di appoggio */
  int primo, secondo, appoggio, da_copiare;

  primo = iniziale;
  secondo = mediano + 1;
  appoggio = iniziale;

  while (primo <= mediano && secondo <= finale)
  {
    if (A[primo] <= A[secondo]) {
      B[appoggio] = A[primo]; primo++;
    }
    else
    {
      B[appoggio] = A[secondo]; secondo++;
    }
    appoggio++;
  }
```

```
  if (secondo > finale)
        /* e' finito prima il secondo sottovettore; copia da A in B tutti
           gli elementi del primo sottovettore fino a mediano */
    for (da_copiare = primo; da_copiare <= mediano; da_copiare++) {
      B[appoggio] = A[da_copiare];
      appoggio++;
    }
  else
        /* e' finito prima il primo sottovettore
        copia da A in B tutti gli elementi del
        secondo sottovettore fino a finale */
    for (da_copiare = secondo; da_copiare <= finale; da_copiare++) {
      B[appoggio] = A[da_copiare];
      appoggio++;
    }

     /* ricopia tutti gli elementi da iniziale a finale da B ad A */
  for (da_copiare = iniziale; da_copiare <= finale; da_copiare++)
    A[da_copiare] = B[da_copiare];
} /* MergeVettore */
```

Esercizio su Array

Siano A e B due array di interi di lunghezza identica nota (p.es. 10).
Ordinare i due array utilizzando il MergeSort e creare un terzo array C contenente le somme dei valori di A e B (ordinati) nel seguente modo:
C[0] = A[0] + B[9]
C[1] = A[1] + B[8]
...
C[9] = A[9] + B[0]

Da dove iniziare? Ragioniamo Top-Down

1. Ordiniamo gli array A e B (già sappiamo come fare)
2. Creiamo l'array C inserendo le somme degli elementi di A e B

Il punto 1 lo risolviamo applicando le funzioni appena studiate...concentriamoci sul punto 2:

- Definiamo il vettore
- Facciamo un ciclo per l'inserimento dei valori nel vettore risultato C
- Per ogni elemento C[i] calcoliamo la somma da inserire

Pseudo Codice

```
PROBLEMA(A,B)
   Merge-Sort(A,10)
   Merge-Sort(B,10)

   per ogni elemento di C
       C[i] = somma elementi di A e B

   restituisco C
```

Come faccio a inserire in un array di 10 elementi 10 valori?

```
   FOR I = 1 TO 10
      C[I] = somma elementi di A e B

   restituisco C
```

Somma degli elementi: I e 11-I

```
      C[I] = A[I] + B[11-I]
```

Traduzione in codice C

```
#include <stdio.h>
#define NumElementi 10

typedef int TipoElemVettore;
typedef TipoElemVettore TipoVettore[NumElementi];

#include "MergeVettore.c"
#include "MergeSort.c"

void problema(TipoVettore A,TipoVettore B,TipoVettore C)
{
  int i;

  MergeSort(A,10);
  MergeSort(B,10);

  for(i=0; i<NumElementi; i++)
    {
      C[i] = A[i] + B[(9 - i)];
      printf("C[%d]= %d \n",i,C[i]);
    }
}
```

```
main()
{
  TipoVettore A, B, C;
  int i;

  for(i = 0; i < NumElementi; i++)
    {printf("A[%d]=",i);scanf("%d",&A[i]);printf("\n");}

  for(i = 0; i < NumElementi; i++)
    {printf("B[%d]=",i);scanf("%d",&B[i]);printf("\n");}

  problema(A,B,C);
}
```

Esercizio: Date 3 sequenze di caratteri non ordinati (lunghezza fissata identica), contare e memorizzare il numero di occorrenze di ogni lettera dell'alfabeto.
Definire i tipi utilizzati, scrivere una pseudo-codifica dell'algoritmo e darne una possibile codifica in linguaggio C.
Analisi di complessità.