

ALBERI E RICORSIONE

ALGORITMI DI ORDINAMENTO

Decimo Laboratorio

ALGORITMI DI ORDINAMENTO



INTRODUZIONE

- ▶ Ordinare una sequenza di informazioni significa effettuare una permutazione in modo da rispettare una relazione d'ordine tra gli elementi della sequenza (p.e. minore o uguale ovvero non decrescente)
- ▶ Sulla sequenza ordinata diventa più semplice effettuare ricerche



COMPLESSITÀ COMPUTAZIONALE

- ▶ È importante avere un indicatore di confronto tra i vari algoritmi possibili di ordinamento, indipendentemente dalla piattaforma hw/sw e dalla struttura dell'elemento informativo
- ▶ La complessità computazionale si basa sulla valutazione del numero di operazioni elementari necessarie (Confronti, Scambi)
- ▶ Si misura come funzione del numero n di elementi della sequenza
- ▶ Gli algoritmi di ordinamento interno si dividono in
 - ▶ Algoritmi semplici - complessità $O(n^2)$
 - ▶ Algoritmi evoluti - complessità $O(n \cdot \log(n))$



ALGORITMI DI ORDINAMENTO

- ▶ Bubble sort
- ▶ Selection sort
- ▶ Merge sort



BUBBLE SORT

- ▶ Si tratta di un algoritmo semplice.
- ▶ Il nome deriva dalla analogia dei successivi spostamenti che compiono gli elementi dalla posizione di partenza a quella ordinata simile alla risalita delle bolle di aria in un liquido.
- ▶ Il bubbling si può realizzare in molte versioni basate sullo stesso principio.



DESCRIZIONE INFORMALE

- ▶ Si consideri un vettore di n elementi.
- ▶ Si vogliono ordinare gli elementi in ordine non decrescente
- ▶ Facciamo “risalire le bolle dal fondo”
- ▶ Sono necessarie al più $n-1$ scansioni del vettore (iterazioni)
- ▶ Si utilizza un flag “ordinato”, che viene inizializzato a 1 all’inizio di ogni ciclo (ossia si ipotizza il vettore ordinato)
- ▶ Ad ogni ciclo vengono confrontati gli elementi del vettore, e se essi non sono nell’ordine desiderato, vengono scambiati



CODICE C DEL BUBBLE SORT

```
1.  #include <stdio.h>

1.  void scambia(tipobase v[], unsigned long i,
    unsigned long j)
2.  {
3.      tipobase tmp=v[i];
4.      v[i]=v[j];
5.      v[j]=tmp;
6.  }
```

CODICE C DEL BUBBLE SORT (CONT.)

```
1. void BubbleSort(tipobase v[], unsigned long dim)
2. {
3.     short ordinato=0;
4.     unsigned long i,j;
5.     for (j=0; j<dim-1 && !ordinato; j++) {
6.         ordinato=1;
7.         for (i=dim-1; i>j; i--)
8.             if (v[i]<v[i-1]) {
9.                 scambia(v,i,i-1); ordinato=0;
10.            }
11.
12.        }
13. }
```

SIMULAZIONE BUBBLE SORT

Esempio: Sia **A** il vettore da ordinare



Ordinato=1

Vettore Ordinato!



ANALISI DELLE PRESTAZIONI

BUBBLE SORT (1)

- ▶ Nel bubble sort l'operazione dominante è il confronto tra coppie di elementi, l'operazione di scambio degli elementi viene eseguita meno spesso rispetto al confronto degli elementi e ha lo stesso costo.
 - ▶ L'algoritmo BubbleSort esegue un numero di operazioni variabili a seconda dell'ordinamento già presente nel vettore
 - ▶ Il caso migliore si ha quando l'array è già ordinato ed è sufficiente effettuare solo un ciclo per verificare che tutti gli elementi sono ordinati, quindi sono sufficienti $n-1$ confronti.
 - ▶ Il caso peggiore si ha quando gli elementi nell'array sono ordinati in senso decrescente.
-



ANALISI DELLE PRESTAZIONI

BUBBLE SORT (2)

- ▶ **Nel caso peggiore il bubble sort effettua i seguenti confronti**
 - ▶ durante il primo ciclo vengono eseguiti $n-1$ confronti e altrettanti scambi
 - ▶ durante il secondo ciclo vengono eseguiti $n-2$ confronti e altrettanti scambi
 - ▶ durante il terzo ciclo vengono eseguiti $n-3$ confronti e altrettanti scambi
 - ▶ ...
 - ▶ durante l' n -esimo ciclo viene eseguito un confronto e uno scambio
-



ANALISI DELLE PRESTAZIONI

BUBBLE SORT (3)

- ▶ Quindi nel caso peggiore il numero di confronti e scambi su di un array di dimensione n è:

$$\sum (n-i) = n \sum 1 - \sum i = n(n-1) - n(n-1)/2 = n(n-1)/2$$

- ▶ Per cui il tempo di esecuzione dell'algoritmo tende a n^2



SELECTION SORT

- ▶ Sia A un array di n elementi da ordinare. L'algoritmo selection-sort seleziona l'elemento con valore minore e lo scambia con il primo elemento.
 - ▶ Tra gli $n-1$ elementi rimasti viene poi ricercato quello con valore minore e scambiato con il secondo e così di seguito fino all'ultimo elemento.
 - ▶ METODO:
 - ▶ Iteriamo i seguenti passi:
 - ▶ l'array A è diviso in 2 parti
Parte iniziale ordinata | parte da ordinare
 - ▶ cerchiamo l'elemento minimo nella parte non ordinata e lo scambiamo con il primo della parte non ordinata
-



SELECTION SORT

Parte iniziale ordinata | parte da ordinare

▶ **I iterazione:**

- ▶ $A[1..n]$ non ordinato, cerca minimo di
- ▶ $A[1..n]$ e scambialo con $A[1]$. Quindi: $A[1] | A[2..n]$

▶ **II iterazione:**

- ▶ $A[1]$ ordinato, $A[2..n]$ non ordinato, cerca minimo di $A[2..n]$ e scambialo con $A[2]$. Quindi: $A[1]A[2] | A[3..n]$

▶ **generica iterazione:**

- ▶ $A[1..i-1]$ ordinato, $A[i..n]$ non ordinato, cerca minimo di $A[i..n]$ e scambialo con $A[i]$. Quindi: $A[1..i-1] A[i] | A[i+1..n]$

▶ **Per $i=n-1$:**

- ▶ $A[1..n-2] | A[n-1], A[n]$ cerca il minimo tra $A[n-1]$ e $A[n]$, scambialo con $A[n-1]$

ARRAY A ORDINATO!



RIORDINAMENTO CON SELECTION SORT

```
void riordina (int vett[ ], int n) /*
    funzione di riordino */
{
    int min, ind, i, j;
    for (i = 0; i < n-1; i++) /* definisce il vettore in cui
        cercare il min. */
    {
        min = vett[i]; /* inizializzazioni per ogni nuovo
            "sottovettore" */
        ind = i;
        for (j = i + 1; j < n; j++) /* cerca il minimo del
            "sottovettore" */
            if (vett[j] < min)
            {
                min = vett[j];
                ind = j;
            }
        vett[ind] = vett[i]; /* scambia il primo elemento del
            "sottovettore" */
        vett[i] = min; /* con l'elemento minimo
            appena trovato */
    }
}
```

SIMULAZIONE SELECTION SORT

- Sia A un array di 6 elementi da ordinare in modo crescente



ANALISI SELECTION SORT (1)

- ▶ L'operazione dominante per il selection-sort è il confronto tra coppie di elementi, per $n-1$ volte bisogna determinare il minimo tra gli elementi non ordinati e se necessario effettuare uno scambio:
 - ▶ Durante la prima iterazione bisogna effettuare $n-1$ confronti per determinare il minimo tra gli n elementi.
 - ▶ Durante la seconda iterazione bisogna effettuare $n-2$ confronti per determinare il minimo tra gli $n-1$ elementi.
 - ▶
-



ANALISI SELECTION SORT (2)

- ▶ Durante la $n-1$ iterazione bisogna effettuare 1 confronti per determinare il minimo tra 2 elementi.
- ▶ Quindi il numero di confronti è

$$\sum_{n-i} = (n-1) + (n-2) + \dots + 2 + 1 = n(n-1)/2$$

- ▶ Per cui il tempo di esecuzione dell'algoritmo tende a n^2
 - ▶ Il comportamento del selection-sort è indipendente dall'ordine preesistente nell'array da ordinare: il caso peggiore (array ordinato in ordine decrescente) e quello migliore coincidono (array ordinato in ordine crescente)
-



ALGORITMI DI ORDINAMENTO RICORSIVI

- ▶ ***“DIVIDE ET IMPERA”*** dicevano i latini: sia con il significato che una volta conquistato un nuovo territorio è meglio tenere diviso il popolo, se è ostile, per evitare rivolte e sovversioni; sia con il significato di partizionare il territorio per amministrarlo meglio.
- ▶ Il motto latino ‘Divide et Impera’ è stato acquisito dall’informatica come tecnica per risolvere problemi complessi, in pratica si genera una sequenza di istanze via via più semplici del problema, fino all'istanza che non è più suddivisibile e che ha soluzione banale



MERGE SORT

(ORDINAMENTO PER FUSIONE)

- ▶ Il Merge-sort è formulato ricorsivamente (*divide et impera*).
 - ▶ In pratica l'algoritmo Merge-sort divide ripetutamente il vettore di input a metà, finché si generano segmenti contenenti uno o due componenti, poi ciascun segmento viene ordinato e si effettua la fusione fra i segmenti in sequenza inversa rispetto alla fase di divisione:
 - ▶ Se l'*array* ha due elementi, essi vengono confrontati e scambiati se non sono ordinati
 - ▶ Se l'*array* ha più di due elementi lo si divide a metà (*'divide'*) e si ordinano i due *array* ottenuti usando lo stesso metodo ricorsivamente (*'impera'*). Infine gli *array* vengono *fusi* per formare l'*array* ordinato (*fusione*)
-



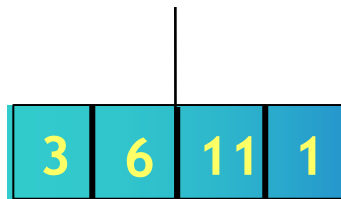
SIMULAZIONE

Dividi a metà!

A



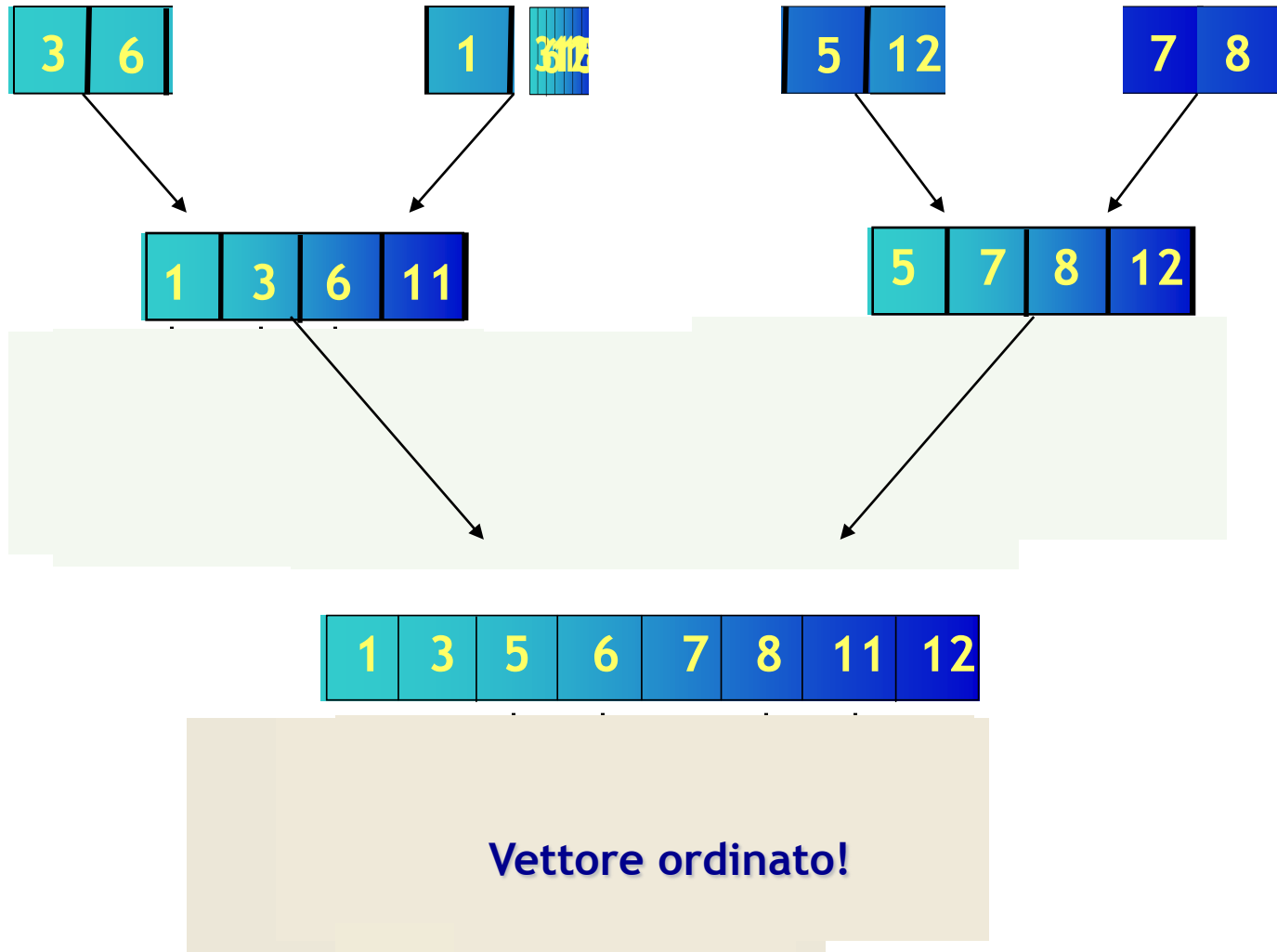
Dividi a metà
ricorsivamente!



Ogni array viene ordinato con un semplice confronto
tra i suoi due elementi!



ESEMPIO: FUSIONE MERGE



ALGORITMO MERGESORT

```
void MergeSort(int A[], int p, int r) {  
    int q;  
  
    if (p < r) {  
        q = (p+r)/2;  
        MergeSort(A, p, q);  
        MergeSort(A, q+1, r);  
        Merge(A, p, q, r);  
    }  
    return;  
}
```



LA FUNZIONE MERGE

```
void Merge(int A[], int p, int q, int r) {
    int i, j, k, B[MAX];

    i = p;    j = q+1;    k = 0;
    while (i<=q && j<=r) {
        if (A[i]<A[j]) {
            B[k] = A[i];    i++;
        } else {
            B[k] = A[j];    j++;    }
        k++;    }
    while (i<=q) {
        B[k] = A[i];    i++;    k++;    }
    while (j<=r) {
        B[k] = A[j];    j++;    k++;    }
    for (k=p; k<=r; k++)
        A[k] = B[k-p];
    return;}
```



COMPLESSITÀ MERGE SORT

- ▶ L'operazione dominante del merge-sort è la fusione di sottosequenze ordinate, il costo di questo algoritmo di ordinamento dipende quindi da quante volte viene eseguita la fusione e da quanto costa ciascuna operazione di fusione
 - ▶ Analizziamo il tempo di esecuzione:
 - ▶ Sia $M(n)$ il numero di confronti, nel caso peggiore, che MergeSort opera con un input di dimensione n .
 - ▶ Sia $T(n)$ una funzione per il tempo di esecuzione dell'algoritmo, tale che $M(n) \leq T(n)$
 - ▶ $T(n) = 0$, se $n=1$ questo è il caso base (se c'è un solo elemento nell'array non si spende tempo)
-



COMPLESSITÀ MERGE SORT

- ▶ Nel caso $n \geq 2$, il numero dei confronti fatti per ordinare n elementi e' al più uguale alla somma del numero di confronti fatti nel caso peggiore per ciascuno dei seguenti passi dell'algoritmo:
 - ▶ Ordina la parte sinistra dell'array
 - ▶ Ordina la parte destra dell'array
 - ▶ Unisci (merge) le due parti in un array ordinato
- ▶ Quindi

$$T(n) = T(n/2) + T(n/2) + n, \text{ se } n \geq 2$$



OSSERVAZIONI

Il **primo termine** ($T(n/2)$), fissa un limite superiore al numero di confronti necessari ad ordinare la parte sinistra dell'array, che consiste della metà degli elementi dell'intero array.

- ▶ Il **secondo termine** ($T(n/2)$), fissa un limite superiore al numero di confronti necessari ad ordinare la parte destra dell'array.
- ▶ L'**ultimo termine** (n), fissa un limite superiore ai confronti necessari a comporre i due array ordinati di n componenti



ESEMPIO: $N = 8$

- Dobbiamo scomporre la relazione fintanto che non arriviamo al passo base:

$$T(8) = 2T(4) + 8$$

$$T(4) = 2T(2) + 4$$

$$T(2) = 2T(1) + 2$$

$$T(1) = 0$$

- A questo punto possiamo fare le opportune sostituzioni come segue:

$$T(1) = 0$$

$$T(2) = 2T(1) + 2 = 0 + 2$$

$$T(4) = 2T(2) + 4 = 4 + 4 = 8$$

$$T(8) = 2T(4) + 8 = 16 + 8 = 24$$

- In questo caso, il merge-sort richiede al piu' 24 confronti per ordinare 8 elementi, $M(8) \leq 24$
-



DIMOSTRAZIONE (1)

- ▶ Dimostriamo che per n in generale risulta

$$M(n) \leq n \cdot \log n$$

- ▶ Infatti: dall'equazione

$$T(n) = 2T(n/2) + n, \quad n \geq 2$$

$$T(1) = 0$$

- ▶ Per semplificare consideriamo $n=2^k$, ($k=\log_2 n$)

$$\begin{aligned} T(2^k) &= 2T(2^{k-1}) + 2^k = 4T(2^{k-2}) + 2 \cdot 2^k = 8T(2^{k-3}) + 3 \cdot 2^k \\ &= \dots = 2^i T(2^{k-i}) + i \cdot 2^k \end{aligned}$$

- ▶ Quando $i=\log_2 n$ si ha:

$$T(n) = 2^{\log_2 n} T(1) + n \cdot \log n = 0 + n \cdot \log n$$



DIMOSTRAZIONE (2)

- ▶ Quindi il merge-sort richiede un tempo di esecuzione di al più $n \log n$, se n è la lunghezza dell'array da ordinare.
- ▶ Il merge-sort è, per array di grandi dimensioni, è il più efficiente algoritmo di ordinamento basato sui confronti, perché anche nel caso pessimo ha un tempo di esecuzione di al più $n \log n$ e non esistono algoritmi di ordinamento di array basati sui confronti che hanno un tempo di esecuzione inferiore.



RIORDINAMENTO ALFABETICO

- ▶ Realizzare un programma che legge da tastiera una sequenza di nomi (10), li riordina in ordine alfabetico mediante l'algoritmo di *bubble-sort* e infine li visualizza.
- ▶ Analisi:
 - ▶ il problema è identico a quelli visti ma occorre considerare il fatto che i nomi sono delle stringhe;
 - ▶ bisogna pertanto modificare la funzione *bubble* nella definizione, quando esegue il confronto fra i due elementi del vettore e quando scambia fra di loro gli elementi.

RIORDINAMENTO ALFABETICO

```
#include <stdio.h>
```

```
#define NELEM 10
```

```
#define LUNGMAX 50
```

```
#define FALSO 0
```

```
#define VERO 1
```

```
/*    Funzione bubble: riordina un vettore di  
    stringhe lungo n    */
```

```
void bubble (char vett[ ][LUNGMAX], int n)
```

```
/*    prototipo    */
```

RIORDINAMENTO ALFABETICO

```
main( ){
char parole [NELEM][LUNGMAX];
int ind;

printf ("Introduci le parole\n");
for (ind = 0; ind < NELEM; ind++)
    {
        /* Riempie
        il vettore con NELEM parole */
        printf ("\nParola di indice %d: ", ind);
        scanf ("%s", parole[ind]);
    }
bubble (parole, NELEM);
/* Riordina il vettore */
printf ("\n Vettore riordinato \n"); /* Visualizza
il vettore riordinato */
for (ind = 0; ind < ind; ind++)
    printf ("%s\n", parole[ind]);}
```

RIORDINAMENTO ALFABETICO

```
void bubble(char vett[ ][LUNGMAX], int n){

int i, inordine;      char provv[LUNGMAX];
inordine = FALSO;      /*
    inizializza inordine a FALSO */
while (!inordine)      /* Finché il
    vettore non è ordinato */
{
    inordine = VERO; /* ipotizza che sia ordinato,
    cioè di aver finito */
    for (i = 0; i < (n - 1); i++) /* i va dal primo
    al penultimo elem. */
        if (strcmp (vett[i], vett[i+1]) > 0) /* se
    quello che segue è > */
        {
            strcpy (provv, vett[i]); /* scambia i
    due elementi tra loro */
            strcpy (vett[i], vett[i+1]);
            strcpy (vett[i+1], provv);
            inordine = FALSO; } /* è stato
    fatto uno scambio! */
}
```

► 35

ALBERI E RICORSIONE



ALBERI ORDINATI ISOMORFI

- ▶ Due alberi binari ordinati si dicono isomorfi solo se sono identici, cioè o entrambi gli alberi sono vuoti oppure: le radici sono uguali, i rispettivi sottoalberi sinistri sono identici ed i rispettivi sottoalberi destri sono identici.



VERIFICA ISOMORFISMO

- ▶ Scrivere un programma che verifichi se due alberi ordinati sono isomorfi
- ▶ Schema ricorsivo
 - ▶ caso base
 - ▶ caso ricorsivo



VERIFICA ISOMORFISMO

- ▶ Scrivere un programma che verifichi se due alberi ordinati sono isomorfi. Ritorna 0 se non lo sono, 1 altrimenti
 - ▶ Schema ricorsivo
 - ▶ caso base:
 - ▶ $I([], []) = 1$
 - ▶ $I([a|Al|Ar], []) = 0$
 - ▶ $I([], [b|Bl|Br]) = 0$
 - ▶ $I([a|Al|Ar], [b|Bl|Br]) = 0$ // se $a \neq b$
 - ▶ caso ricorsivo
 - ▶ $I([a|Al|Ar], [b|Bl|Br]) = I(Al, Bl) \text{ and } I(Ar, Br)$ // se $a = b$
-



VERIFICA ISOMORFISMO

```
int uguale(Albero A, Albero B)
{ // VERIFICA SE DUE ALBERI ORDINATI SONO
  ISOMORFI

  if (A==NULL || B==NULL)
    return (A==NULL) && (B==NULL);
    // due alberi sono uguali se sono
    entrambi NULL

  return (A->val==B->val) && uguale(A->left,B-
    >left) && uguale(A->right,B->right);
}
```



ALBERI NON ORDINATI ISOMORFI

- ▶ Per gli alberi non ordinati, è del tutto indifferente se una chiave è presente nel sottoalbero destro o nel sottoalbero sinistro della radice. Pertanto due alberi non ordinati sono isomorfi se le loro radici contengono lo stesso elemento ed inoltre o accade che i due sottoalberi sinistri ed i due sottoalberi destri sono isomorfi tra loro oppure accade che il sottoalbero sinistro del primo sia isomorfo al sottoalbero destro del secondo ed il sottoalbero destro del primo è isomorfo al sottoalbero sinistro del secondo.
 - ▶ Una procedura ricorsiva, che sfrutta questa definizione permette di verificare se due alberi A e B, non ordinati, sono isomorfi tra di loro.
-



VERIFICA ISOMORFISMO

- ▶ Scrivere un programma che verifichi se due alberi non ordinati sono isomorfi. Ritorna 0 se non lo sono, 1 altrimenti



VERIFICA ISOMORFISMO

```
int uguale(Albero A, Albero B)
{ // VERIFICA SE DUE ALBERI NON ORDINATI SONO
  ISOMORFI

  if (A==NULL || B==NULL)
    return (A==NULL) && (B==NULL);
    // due alberi sono uguali se sono entrambi
    NULL

  return (A->val==B->val) && ((uguale(A->left,B-
    >left) && uguale(A->right,B->right))

    || (uguale(A->right,B->left) && uguale(A-
    >left,B->right))) );
}
```



VERIFICA SOTTOALBERO

- ▶ Dati due alberi ordinati A e B , verificare se B è sottoalbero di A
- ▶ Definizione di Sottoalbero
 - ▶ dato un albero T , un sottoalbero è costituito da un generico nodo n di T e da tutti i suoi discendenti.
 - ▶ il nodo n è la radice del sottoalbero.
- ▶ Suggerimento: utilizzare la funzione per determinare se due alberi ordinati sono uguali.



VERIFICA SOTTOALBERO

► Schema ricorsivo

► caso base:

- $S([], []) = 1$
- $S([], B) = 0;$
- $S(A, []) = 1;$
- $S([a|Al|Ar], [b|Bl|Br]) = 1$ // se $a = b$ and $I([a|Al|Ar], [b|Bl|Br]) = 1$

► caso ricorsivo:

- $S([a|Al|Ar], [b|Bl|Br]) = S(Al, [b|Bl|Br])$ // se $(b < a)$ o $(a=b$ and
// $I([a|Al|Ar], [b|Bl|Br]) = 0)$
- $S([a|Al|Ar], [b|Bl|Br]) = S(Ar, [b|Bl|Br])$ // se $b > a$



VERIFICA SOTTOALBERO

```
int sottoalbero(Albero A1, Albero B1)    {
    // CERCA SE IN A C'E' LA CHIAVE DELLA ROOT DI B1
    // SE LA TROVA VERIFICA CHE I DUE SOTTO ALBERI SIANO UGUALI
    if(B1==NULL) return 1;
    if (A1!=NULL) {
        if ((A1->val)==(B1->val)) {
            if (uguale(A1,B1) || sottoalbero (A1->left,B1))
                return 1;
        }else{
            if (A1->val>B1->val)
                return sottoalbero (A1->left,B1);
            else
                return sottoalbero (A1->right,B1);
        }else
            return 0;
    }
}
```



VALORE MASSIMO

- ▶ Scrivere una procedura tale che assegnato un albero binario non ordinato di interi positivi restituisca il puntatore ad uno dei nodi contenenti il valore massimo.
- ▶ Schema ricorsivo
 - ▶ caso base:
 - ▶ caso ricorsivo:



VALORE MASSIMO

- ▶ Scrivere una procedura tale che assegnato un albero binario non ordinato di interi positivi restituisca il puntatore ad uno dei nodi contenenti il valore massimo.
 - ▶ Schema ricorsivo
 - ▶ caso base: $M([]) = \text{NULL}$
 - ▶ caso ricorsivo: $M([a|Al, Ar]) = \max(a, M(Al), M(Ar))$
-



VALORE MASSIMO

```
nodo* CercaMax (Albero ANonOrd)
{
    if (AnonOrd==NULL)
        return NULL;
    nodo* max=AnonOrd;
    nodo* Msx=CercaMax (ANonOrd->left) ;
    If (Msx && Msx->val>max->val)
        max=Msx;
    nodo* Mdx=CercaMax (ANonOrd->right) ;
    if (Mdx && Mdx->val>max->val)
        max=Mdx;
    Return max;
}
```



NUMERI COMPRESI

- ▶ Scrivere una funzione che assegnato un albero binario di interi positivi non ordinato e due numeri positivi $N1$ e $N2$ restituisca la quantità di numeri pari compresi tra $N1$ e $N2$.
- ▶ Schema ricorsivo
 - ▶ caso base:
 - ▶ caso ricorsivo:



NUMERI COMPRESI

- ▶ Scrivere una funzione che assegnato un albero binario di interi positivi non ordinato e due numeri positivi $N1$ e $N2$ restituisca la quantità di numeri pari compresi tra $N1$ e $N2$.
- ▶ Schema ricorsivo
 - ▶ caso base: $N([]) = 0$
 - ▶ caso ricorsivo:
 - ▶ $N([a|Al|ar]) = 1 + N(Al) + N(Ar)$ // se $N1 < a < N2$
 - ▶ $N([a|Al|ar]) = 0 + N(Al) + N(Ar)$ // altrimenti



NUMERI COMPRESI

```
int Elab_Albero (Albero ANonOrd, int N1a,
    Albero N2a)
{
    if (ANonOrd==NULL)
        return 0;
    int cont=0;
    if ( (ANonOrd->val%2==0) && (ANonOrd->val>N1a) && (ANonOrd->val<N2a) )
        cont =1;
    return cont + Elab_Albero (ANonOrd->left, N1a, N2a) +
        Elab_Albero (ANonOrd->right, N1a, N2a);
}
```



ESERCIZIO: NODI LIVELLO PARI

- ▶ Scrivere una funzione che dato un albero in input inserisca in una lista solo i nodi di livello pari
- ▶ Schema ricorsivo:
 - ▶ caso base:
 - ▶ caso ricorsivo:



ESERCIZIO: NODI LIVELLO PARI

- ▶ Scrivere una funzione che dato un albero in input inserisca in una lista solo i nodi di livello pari
 - ▶ Schema ricorsivo:
 - ▶ $[a, Al, Ar]$: albero di radice a , figli Al e Ar .
 - ▶ L : lista
 - ▶ n : livello del nodo “ a ”
 - ▶ caso base: $I([], L, n) = L$
 - ▶ caso ricorsivo:
 - ▶ $I([a, Al, Ar], L, n) = \begin{cases} [I(Al, L, n+1)|I(Ar, L, n+1)|a] & // \text{ se } n \text{ è pari} \\ [I(Al, L, n+1)|I(Ar, L, n+1)] & // \text{ se } n \text{ è dispari} \end{cases}$
-



SOLUZIONE (1)

```
lista salvaPari (Albero A, lista L, int n) {  
    if (A == NULL) return L;  
    if (n % 2 == 0)  
        L = inserisciTesta (L, A->value);  
    L = salvaPari (A->left, L, n + 1);  
    L = salvaPari (A->right, L, n + 1);  
    return L;  
}
```



SOLUZIONE (2)

```
lista salvaPari (Albero A, int n) {
    if (A == NULL) return NULL;
    lista Lhead = NULL;
    Lhead = salvaPari(A->left, n + 1);
    if (Lhead != NULL) {
        lista Ltail = Lhead;
        while (Ltail->next != NULL) Ltail = Ltail->next;
        Ltail->next = salvaPari(A->right, n + 1);
    } else Lhead = salvaPari(A->right, n + 1);

    if (n % 2 == 0)
        Lhead = inserisciListaTesta(Lhead, A->val);

    return Lhead;
}
```



SOLUZIONE (3)

```
lista salvaPari(Albero A, lista L){
    if(A == NULL) return L;
    L = inserisciListaTesta(L, A->val);

    if(A->left != NULL){ // CONTROLLO IMPORTANTE!!!
        L = salvaPari(A->left->right, L);
        L = salvaPari(A->left->left, L);
    }

    if(A->right != NULL){ // CONTROLLO IMPORTANTE!!!
        L = salvaPari(A->right->right, L);
        L = salvaPari(A->right->left, L);
    }

    return L;
}
```



ESERCIZIO: CONTA NODI SINISTRI...

- ▶ Scrivere una funzione che dato un albero in input Conti quanti sono i nodi sinistri che hanno come unico figlio una foglia.
- ▶ Schema Ricorsivo
 - ▶ caso base:
 - ▶ caso ricorsivo:



ESERCIZIO: CONTA NODI SINISTRI...

- ▶ Scrivere una funzione che dato un albero in input Conti quanti sono i nodi sinistri che hanno come unico figlio una foglia.
- ▶ Schema Ricorsivo
 - ▶ caso base: $C([]) = 0$
 - ▶ caso base: $C([a, Al, Ar]) = 1$ // se $Al == \text{NULL}$ && Ar è
// foglia o viceversa
 - ▶ caso ricorsivo: $C([a, Al, Ar]) = C(Al)$



SOLUZIONE

```
int conta(Albero A) {
    if(A == NULL) return 0;

    Albero left = A->left;
    Albero right = A->right;

    if(left == NULL && right != NULL && right->left ==
        NULL && right->right == NULL) return 1;

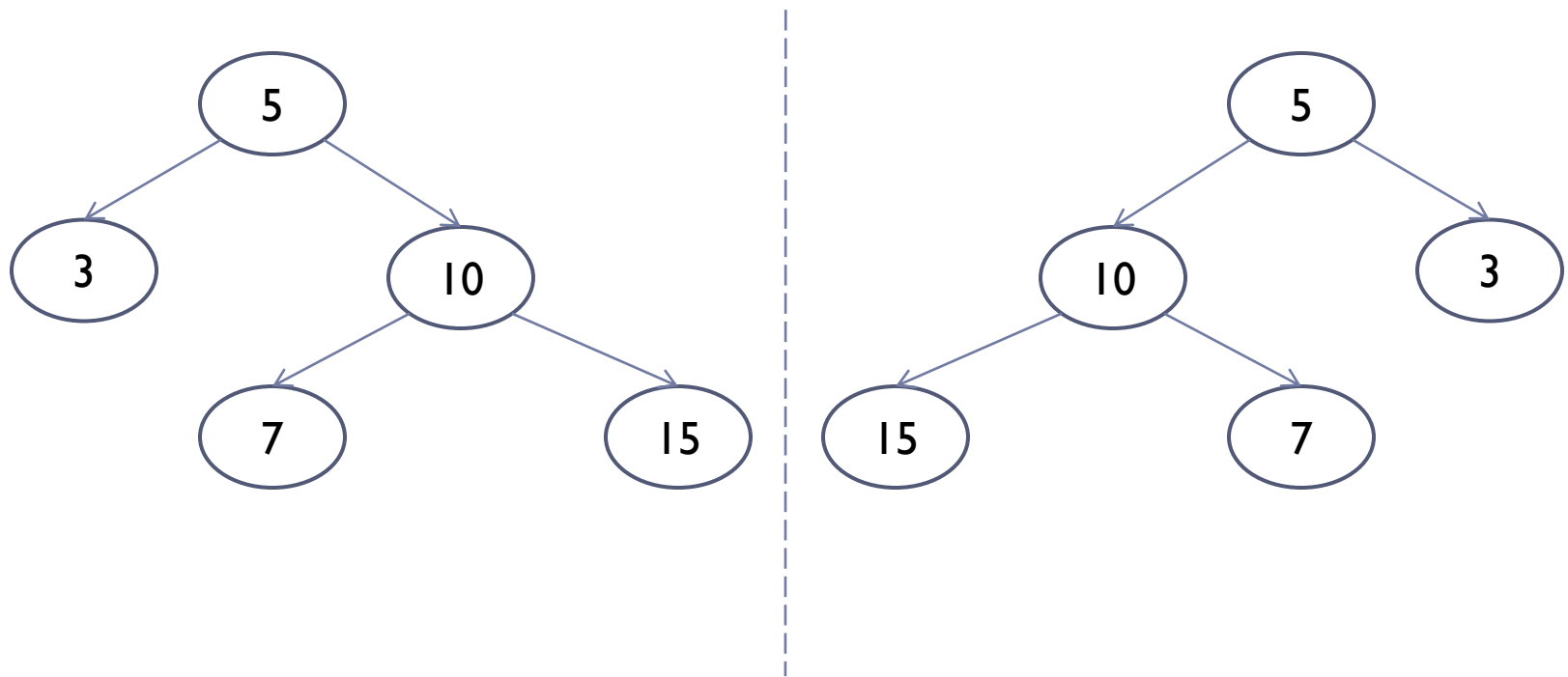
    if(right == NULL && left != NULL && left->left ==
        NULL && left->right == NULL) return 1;

    if(A->right != NULL)
        return conta(A->left) + conta(A->right->left);
    return conta(A->left);
}
```



ESERCIZIO: SPECCHIO

- Scrivere un programma che ritorni un albero “specchio” di quello ricevuto in input.
- Esempio:



ESERCIZIO: SPECCHIO

- ▶ Scrivere un programma che ritorni un albero “specchio” di quello ricevuto in input.
- ▶ Schema ricorsivo
 - ▶ caso base: $S([]) = []$
 - ▶ caso ricorsivo: $S([a, Al, Ar]) = [a, S(Ar), S(Al)]$



SOLUZIONE

```
albero specchio (Albero Ao) {  
    if (Ao == NULL) return NULL;  
  
    albero aux = NULL  
    aux = (nodo*) malloc(sizeof(nodo));  
    aux->value = Ao->value;  
    aux->left = specchio(Ao->right);  
    aux->right = specchio(Ao->left);  
  
    return aux;  
}
```



ESERCIZIO: CALCOLA NODI PER LIVELLO

- ▶ Scrivere un programma che salva in una lista ordinata in modo crescente, quanti nodi ci sono per ogni livello.
- ▶ Schema Ricorsivo
 - ▶ caso base:
 - ▶ caso ricorsivo:



ESERCIZIO: CALCOLA NODI PER LIVELLO

- ▶ Scrivere un programma che salva in una lista ordinata in modo crescente, quanti nodi ci sono per ogni livello.
- ▶ Schema Ricorsivo
 - ▶ caso base: $C([], L) = L$
 - ▶ caso ricorsivo:
 - ▶ $C([a, Al, Ar], L) = [L_{i-1}, Li + 1 \mid C(Al, [L_{i+1} \mid L_{i+2}..]), C(Ar, [L_{i+1} \mid L_{i+2}..])]$



ESERCIZIO: CALCOLA NODI PER LIVELLO

```
lista CalcolaNodiLivello(Albero al, lista l) {
    if(!al) return l;

    if(l == NULL) {
        l = (nodo*) malloc(sizeof(nodo));
        l->val = 0; l->next = NULL;
    }
    l->val++;
    l->next = CalcolaNodiLivello(al->left, l->next);
    l->next = CalcolaNodiLivello(al->right, l->next);
    return l;
}
```



ESERCIZIO: AGGIUNGI NODO

- ▶ Dato un albero binario i cui nodi contengono interi, si vuole scrivere una funzione che aggiunga ad ogni foglia un figlio contenente la somma dei valori che appaiono nel cammino dalla radice a tale foglia.
- ▶ Schema Ricorsivo:
 - ▶ caso base:
 - ▶ caso ricorsivo:



ESERCIZIO: AGGIUNGI NODO

- ▶ Dato un albero binario i cui nodi contengono interi, si vuole scrivere una funzione che aggiunga ad ogni foglia un figlio sinistro contenente la somma dei valori che appaiono nel cammino dalla radice a tale foglia.
- ▶ Schema Ricorsivo:
 - ▶ caso base: $S([a, [], []], \text{sum}) = [a, a + \text{sum}, []]$
 - ▶ caso ricorsivo: $S([a, Al, Ar], \text{sum}) = [a, S(Al, \text{sum} + a), S(Ar, \text{sum} + a)]$



ESERCIZIO: AGGIUNGI NODO

```
albero AggiungiNodo(Albero al, int sum){
    if(al == NULL) return NULL;

    if(al->left == NULL && al->right == NULL){
        albero tmp = (nodo*) malloc(sizeof(nodo));
        tmp->val = al->val + sum;
        tmp->left = NULL; tmp->right = NULL;
        al->left = tmp;
        return al;
    }

    sum += al->val;
    al->left = AggiungiNodo(al->left, sum);
    al->right = AggiungiNodo(al->right, sum);
    return al;
}
```



CONTA FOGLIE

- ▶ Scrivere un programma che conti il numero di foglie contenute nell'albero
- ▶ Una foglia è un particolare tipo di nodo dell'albero, avente entrambi i figli vuoti...
- ▶ Schema ricorsivo
 - ▶ caso base:
 - ▶ caso ricorsivo:



CONTA FOGLIE

- ▶ Scrivere un programma che conti il numero di foglie contenute nell'albero
 - ▶ Una foglia è un particolare tipo di nodo dell'albero, avente entrambi i figli vuoti...
 - ▶ Schema ricorsivo
 - ▶ caso base: $C([]) = 0$
 - ▶ caso base: $C([a|Al|Ar]) = 1$; //se $Al = Ar = \text{NULL}$
 - ▶ caso ricorsivo: $C([a|Ad|As]) = C([Ad]) + C([As])$ // altrimenti
-



CONTA FOGLIE

```
int contaFoglie(Albero al) {  
    if(!al) return 0;  
    if(al->left == NULL && al->right == NULL)  
        return 1;  
    return contaFoglie(al->left) +  
        contaFoglie(al->right);  
}
```



TROVA ALTEZZA

- ▶ Scrivere un programma che ritorni l'altezza dell'albero, ove l'altezza è definita come il livello del nodo di livello massimo.
- ▶ Schema Ricorsivo
 - ▶ caso base: $H([]) = 0$
 - ▶ caso ricorsivo: $H([a|Ad|As]) = 1 + \max(H([Ad]), H([As]));$



TROVA ALTEZZA

```
int trovaAltezza(Albero al) {  
    if(!al) return 0;  
    int left = trovaAltezza(al->left);  
    int right = trovaAltezza(al->right);  
    if(left < right) return 1 + right;  
    return 1 + left;  
}
```



INSIEMI E RICORSIONE



- ▶ *Lista con due proprietà*
 - ▶ *Ordinata*
 - ▶ *Senza repliche*

```
typedef Lista Insieme;
```

- ▶ *Ogni operazione che viene eseguita su un oggetto di tipo insieme, deve rispettare le due proprietà.*

FUNZIONE INSERIMENTO

- ▶ *L'inserimento di un nuovo elemento all'interno di un insieme, richiede di individuare la corretta posizione di inserimento, per mantenere la proprietà di ordinamento, e dovrà controllare che l'elemento non sia già presente.*
 - ▶ *Si può fare sia ricorsivamente sia iterativamente*
 - ▶ Schema Ricorsivo
 - ▶ caso base: $I([], v) = [v]$
 - ▶ caso base: $I([a|L], v) = \begin{cases} [v|a|L] & \text{se } a > v \\ [a|L] & \text{se } a = v \end{cases}$
 - ▶ caso ricorsivo: $I([a|L], v) = [a|I(L, v)]$ se $a < v$
-



FUNZIONE INSERIMENTO

```
Insieme InsiemeIns(Insieme li, int el) {  
    if (!li || li->val > el)  
        return inserisciListaTesta(li, el);  
    if (li->val==el) return li;  
  
    if (li->val<el) {  
        Insieme pi=InsiemeIns(li->next, el);  
        li->next=pi;  
        return li;  
    }  
}
```



ESERCIZIO: INTERSEZIONE

- ▶ *Scrivere la funzione che esegue l'intersezione di due insiemi.*

```
Insieme InsiemeIntersezione(Insieme l1, Insieme l2);
```

- ▶ SCHEMA RICORSIVO:

- ▶ Caso base:

- ▶ Caso generale:



ESERCIZIO: INTERSEZIONE

- ▶ Scrivere la funzione che esegue l'intersezione di due insiemi.

Insieme InsiemeIntersezione(Insieme l1, Insieme l2);

- ▶ SCHEMA RICORSIVO:

- ▶ Caso base:
$$\begin{cases} I([],[]) = [] \\ I([], [a|L]) = [] \\ I([a|L], []) = [] \end{cases}$$

- ▶ Caso generale:
$$I([a|L1], [b|L2]) = \begin{cases} I(L1, [b|L2]) & \text{se } a < b \\ I([a|L1], L2) & \text{se } a > b \\ [a|I(L1, L2)] & \text{se } a = b \end{cases}$$



ESERCIZIO: INTERSEZIONE

```
Insieme InsiemeIntersezione(Insieme l1, Insieme l2) {  
  
    if (!l1 && !l2) return NULL;  
    if (l1 && !l2) return InsiemeIntersezione(l1->next, l2);  
    if (!l1 && l2) return InsiemeIntersezione(l1, l2->next);  
  
    if (l1->val==l2->val)  
        return ListInsInTesta(InsiemeIntersezione(l1->next,  
l2->next), l1->val);  
  
    if (l1->val<l2->val)  
        return InsiemeIntersezione(l1->next, l2);  
  
    if (l1->val>l2->val)  
        return InsiemeIntersezione(l1, l2->next);  
}
```



ESERCIZIO: DIFFERENZA

- *Scrivere la funzione che esegue la differenza tra due insiemi.*

Insieme InsiemeDifferenza(Insieme l1, Insieme l2);

- SCHEMA RICORSIVO:

- Caso base:
$$\begin{cases} D([],[])=[] \\ D([],[a|L])=[] \\ D([a|L],[])=[a|D(L,[])] \end{cases}$$

- Caso generale:
$$D([a|L1],[b|L2]) = \begin{cases} [a|D(L1,[b|L2])] & \text{se } a < b \\ D([a|L1],L2) & \text{se } a > b \\ D(L1,L2) & \text{se } a = b \end{cases}$$



ESERCIZIO: DIFFERENZA

```
Insieme InsiemeDifferenza(Insieme l1, Insieme l2) {  
    if (!l1 && !l2) return NULL;  
    if (l1 && !l2)  
        return ListInsInTesta(InsiemeDifferenza(l1->next, l2),  
l1->val);  
    if (!l1 && l2) return InsiemeDifferenza(l1, l2->next);  
  
    if (l1->val==l2->val)  
        return InsiemeDifferenza(l1->next, l2->next);  
  
    if (l1->val<l2->val)  
        return ListInsInTesta(InsiemeDifferenza(l1->next, l2),  
l1->val);  
  
    if (l1->val>l2->val)  
        return InsiemeDifferenza(l1, l2->next);  
}
```



SETTIMA CONSEGNA



SETTIMA CONSEGNA

- ▶ Consegna dovrà essere eseguita entro il giorno **5 febbraio 2012**
- ▶ Le consegne effettuate successivamente non verranno considerate...
- ▶ Comando per la consegna:
consegna consegna7



ESERCIZIO:

IL GIOCO DI TOM E JERRY

- ▶ Tom e Jerry hanno a disposizione un albero binario di ricerca di interi a testa.
- ▶ I due muovono a turni.
- ▶ Inizia Jerry a muovere.
- ▶ Scopo del gioco per TOM
 - ▶ acciuffare JERRY. Il che si concretizza per TOM nel trovarsi in un nodo il cui valore è identico a quello in cui si trova Jerry.



STRATEGIA DEL GIOCO

- ▶ Ogni giocatore può soltanto muovere da padre a figlio e non viceversa.
 - ▶ Se un giocatore raggiunge una cella che ha un solo nodo figlio può soltanto muovere verso quel nodo figlio.
 - ▶ Se un giocatore raggiunge un nodo che ha due figli allora si comporta come segue:
 - ▶ se Jerry si trova in un nodo con valore “maggiore” a quello di Tom, Jerry cerca di scappare da Tom muovendo a destra;
 - ▶ se il nodo ha invece valore minore di quello di Tom, allora Jerry muove a sinistra Tom farà l’opposto cercando di avvicinarsi a Jerry. In pratica, se Tom si trova in un nodo con valore “maggiore” a quello di Jerry, Tom muove a sinistra. Se invece è “minore”, allora muove a destra.
 - ▶ se un giocatore raggiunge una foglia non può più muovere.
- Condizioni di terminazione
-



STRATEGIA DEL GIOCO

- ▶ Se Tom vince (entrambi i giocatori sono sullo stesso numero), si provvede a rimuovere il nodo dall'albero di Jerry e il gioco termina.
 - ▶ Se entrambi i giocatori raggiungono una foglia e Tom non ha vinto, si trasferisce la foglia raggiunta da Jerry nell'albero di Tom e si riparte con il gioco.
 - ▶ Quest'ultima operazione è ammessa per al più un numero prefissato di volte.
 - ▶ Si consideri per semplicità che tutti i nodi dell'albero abbiano valore differente. Per cui, l'operazione di inserimento nell'albero di Tom non ha 'effetto' se il nodo è già presente.
-



IMPLEMENTAZIONE

- ▶ Scrivere in linguaggio C un menù a scelta multipla che permetta le seguenti funzioni:
 - ▶ Riempimento (casuale o manuale) dei due alberi Tom e Jerry.
 - ▶ Per semplicità si definiscano due costanti RANGE e TOT, e di riempire l'albero con TOT nodi presi nel range 1? RANGE
 - ▶ Simulazione del gioco.
 - ▶ Tale funzione prende in input i due alberi e restituisce i due alberi modificati al termine del gioco e riporta quanti turni sono stati giocati (si supponga che al più possono essere giocati NUM turni)
 - ▶ Stampa in ordine del contenuto degli alberi prima e dopo il gioco



RIPASSO



RICORSIONE



ESERCIZIO: RICORSIONE 1

- ▶ Si calcoli tramite una funzione ricorsiva il numero di cifre in base 10 necessario per rappresentare un numero naturale N immesso dall'utente.



ESERCIZIO: RICORSIONE

- ▶ Si calcoli tramite una funzione ricorsiva il numero di cifre in base 10 necessario per rappresentare un numero naturale N immesso dall'utente.
- ▶ Caso base: $C(N) = 1$ se $N < 10$
- ▶ Ricorsione: $C(N) = 1 + C(N/10)$



SOLUZIONE

```
int C(int N) {  
    if(N<10) return 1;  
    return 1 + C(N/10);  
}
```



ESERCIZIO: RICORSIONE 2

- ▶ Modellare il seguente problema:
 - ▶ individuare, in un gruppo di 10 palline, la pallina di peso maggiore
- ▶ Suggerimento procedimento:
 - ▶ Il peso delle palline sarà memorizzato in un array, riempito da tastiera;
 - ▶ Se il gruppo di palline consiste in una sola pallina, allora essa è banalmente la pallina cercata;
 - ▶ Altrimenti...



ESERCIZIO: RICORSIONE

- ▶ Modellare il seguente problema:
 - ▶ individuare, in un gruppo di 10 palline, la pallina di peso maggiore
- ▶ Suggerimento procedimento:
 - ▶ Il peso delle palline sarà memorizzato in un array, riempito da tastiera;
 - ▶ $\text{max_ric}([]) = []$
 - ▶ $\text{max_ric}([s|L]) = \max(s, \text{max_ric}(L))$



SOLUZIONE

```
int max_ric(int a[], int s, int dim){
    if(dim <= 0) return -1;
    if(s + 1 == dim) return a[s];

    int returned_value = max_ric(a, s+1,
dim);
    if(returned_value < a[s]) return a[s];
    else return returned_value;
}
```



ESERCIZIO: RICORSIONE E ARRAY

- ▶ Dato un insieme di 4 numeri naturali, memorizzati all'interno di un array A , proporre una funzione ricorsiva che stampi in output tutte le possibili permutazioni dell'insieme rappresentato in A .



ESERCIZIO: RICORSIONE E ARRAY

- ▶ Dato un insieme di 4 numeri naturali, memorizzati all'interno di un array A , proporre una procedura ed una ricorsiva che stampi in output tutte le possibili permutazioni dell'insieme rappresentato in A .
- ▶ Esempio con 3:
 - ▶ $S([]) = \text{niente}$
 - ▶ $S([a]) = a$
 - ▶ $S([1,2,3]) = 1 S([2,3]), 2 S([1,3]), 3 S([1,2])$



SOLUZIONE

```
void S(int A[], int i, int dim) {
    int k;
    n = dim - i + 1
    if (n > 1) {
        for (k = n; k < dim; k++) {
            int t = A[i]; A[i] = A[k]; A[k] = t;
        }
        print("%d ", A[i]);
        S(A, i + 1, dim);
    } else {
        print("%d ", A[0]);
    }
}
```



ESERCIZIO: RICORSIONE E MATRICI

- ▶ Scrivere una funzione ricorsiva che restituisca true se la matrice di dimensione 3x3 possiede due righe uguali, false altrimenti.
- ▶ La matrice dovrà essere riempita da tastiera
- ▶ Esempio

$$\text{▶ } M = \begin{pmatrix} 1 & 2 & 3 \\ 3 & 4 & 5 \\ 3 & 2 & 6 \end{pmatrix} \quad \text{SOMMA} = (7 \quad 8 \quad 14)$$



ESERCIZIO: RICORSIONE E MATRICI

- ▶ Scrivere una funzione ricorsiva che calcoli la somma delle righe della matrice di dimensione 5x5. L'output sarà un vettore di dimensione 5...
- ▶ La matrice dovrà essere riempita da tastiera
- ▶ $S([]) = 0$;
- ▶ $S([r|M]) = r + S(M)$;



ESERCIZIO: RICORSIONE E STRINGHE

- ▶ Scrivere una funzione ricorsiva che data una stringa in input e un carattere, ritorni il numero di occorrenze del carattere all'interno della stringa



ESERCIZIO: RICORSIONE E STRINGHE

- ▶ Scrivere una funzione ricorsiva che data una stringa in input e un carattere, ritorni il numero di occorrenze del carattere all'interno della stringa
- ▶ Caso base:
 - ▶ $O([], a) = 0$
- ▶ Ricorsione:
 - ▶ $O([x|s], a) = 1 + O([s], a)$ se $x == a$
 - ▶ $O([x|s], a) = 0 + O([s], a)$ se $x != a$



LISTE E RICORSIONE



ESERCIZIO: STAMPA NODI IN POSIZIONE DISPARI

- ▶ Scrivere un programma che stampa solamente i valori contenuti nei nodi in posizione dispari
- ▶ SCHEMA RICORSIVO:
 - ▶ caso base:
 - ▶ caso generale:



ESERCIZIO: STAMPA NODI IN POSIZIONE DISPARI

- ▶ Scrivere un programma che stampa solamente i valori contenuti nei nodi in posizione dispari
- ▶ SCHEMA RICORSIVO:
 - ▶ caso base: $S([], i) = \text{stampa}[]$
 - ▶ caso generale:
$$S([a|L], i) = \begin{cases} \text{stampa}[a] S([L], i+1) & \text{se } i \text{ è dispari} \\ S([L], i+1) & \text{se } i \text{ è pari} \end{cases}$$



ESERCIZIO: STAMPA NODI IN POSIZIONE DISPARI

```
void stampadispari(lista li, int i) {  
    if(li == NULL) return;  
    if(i % 2 != 0) printf("%d ", li->value);  
    stampadispari(li->next, i + 1);  
}
```



ESERCIZIO: CONTA VALORI

- ▶ *Scrivere una funzione ricorsiva che conta il numero di elementi contenuti in una lista che hanno valore minore di un valore v*

```
int contaMinore(lista li, int v);
```

- ▶ SCHEMA RICORSIVO:

- ▶ caso base:

- ▶ caso generale:



ESERCIZIO: CONTA VALORI

- ▶ *Scrivere una funzione ricorsiva che conta il numero di elementi contenuti in una lista che hanno valore minore di un valore v*

```
int contaMinore(Lista li, int v);
```

- ▶ SCHEMA RICORSIVO:

- ▶ caso base: $C([], v) = 0$

- ▶ caso generale: $C([a|L], v) = \begin{cases} 1+C(L, v) & // \text{ se } a < v \\ 0+C(L, v) & // \text{ se } a \geq v \end{cases}$



ESERCIZIO: CONTA VALORI

```
int contaValori(lista li, int v) {  
    if(li == NULL) return 0;  
    if(li->value == v)  
        return 1 + contaValori(li->next, v);  
    return contaValori(li->next, v);  
}
```



ESERCIZIO: TROVA VALORE

- ▶ *Scrivere una funzione ricorsiva che ritorna 1 se il valore v è contenuto nella lista li , 0 altrimenti.*

```
int trovaValore(lista li, int v);
```

- ▶ SCHEMA RICORSIVO:

- ▶ caso base:

- ▶ caso generale:



ESERCIZIO: TROVA VALORE

- ▶ *Scrivere una funzione ricorsiva che ritorna 1 se il valore v è contenuto nella lista li , 0 altrimenti.*

```
int trovaValore(Lista li, int v);
```

- ▶ SCHEMA RICORSIVO:

- ▶ caso base: $T([], v) = 0$

- ▶ caso generale: $T([a|L], v) = \begin{cases} 1 & // \text{ se } a = v \\ T([L], v) & // \text{ se } a \neq v \end{cases}$



ESERCIZIO: TROVA VALORE

```
int trovaValore(lista li, int v) {  
    if(li == NULL) return 0;  
    if(li->value == v) return 1;  
    return trovaValori(li->next, v);  
}
```

