ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA
DIPARTIMENTO DI
INFORMATICA - SCIENZA E INGEGNERIA

FUNZIONI RICORSIVE

COSIMO LANEVE
cosimo.laneve@unibo.it

CORSO 00819 – PROGRAMMAZIONE

ARGOMENTI [SAVITCH – CAPITOLO 14]

1. ricorsione
2. versione ricorsiva e iterativa di programmi
3. commenti sulla complessità computazionale (efficienza)
4. esempi/esercizi
5. progettazione di algoritmi ricorsivi
6. caso di studio: ricerca binaria in un array

FRASI FAMOSE SULLA RICORSIONE

L. Peter Deutsch (inventore di ghostscript)

to iterate is human, to recurse, divin

Stephen Hawking

to understand recursion one has first to understand recursion

Douglas R. Hofstadter

superational thinkers, by recursive definition, include in their calculation the fact that they are in a group of superational thinkers

Bob Barton

the basic principle of recursive design is to make the parts to have the same power as the whole

VERSO LA RICORSIONE: ESEMPIO

il codice che rimuove i multipli di **n** da una lista **p**

```
ptr_lista multiple_remove(ptr_lista p, int n) {
    ptr_lista p_init = p;
    ptr_lista p_old = p;
    while (p != NULL) {
        if ((p_init == p) && ((p->val) % n == 0)){
                    p_init = p->next ;
                    delete p ;
                    p = p_init ;
                     p_old = p ;
        } else if ((p_init != p) && ((p->val) % n== 0)){
                    p_old->next = p->next ;
                    delete p ;
                    p = p_old->next ;
         } else {
                   p_old = p ;
                   p = p->next ;
         }
    }
    return(p_init);
}
```

VERSO LA RICORSIONE: ESEMPIO

il codice che rimuove i multipli di `n` da una lista `p`

```
ptr_lista multiple_remove_ric(ptr_lista p, int n) {
    if (p == NULL) return(NULL) ;
    else if ((p->val) % n == 0) return (multiple_remove_ric(p->next, n)) ;
    else {  p->next = multiple_remove_ric(p->next, n) ;
            return(p) ;
    }
}
```

DIVIDE ET IMPERA

una tecnica per risolvere i problemi è il

divide et impera

1. si scompone il problema originale in sottoproblemi (**divide**)
2. si risolvono i sottoproblemi
3. dalle soluzioni dei sottoproblemi si deriva la soluzione del problema originale (**impera**)

esempi:

- per studiare le funzioni in analisi: dovete (divide) calcolare le derivate prime per i minimi/massimi, le derivate seconde per i punti di flesso e poi (impera) utilizzare le soluzioni per disegnare il grafo
- preparare 3kg di spaghetti al pesto con pentole da 5 litri

LA RICORSIONE

in molti casi, quando si **divide** un problema in sottoproblemi, questi ultimi sono casi più semplici del problema originale:

- cercare se un numero occorre in un array può essere **diviso** in cercare nella prima metà dell'array e cercare nella seconda metà
- cercare in ciascuna metà è **più semplice** che cercare in tutto l'array
- problemi come questo **possono** essere risolti da funzioni ricorsive

una funzione è ricorsiva quando nel suo corpo c' è un'invocazione a sé stessa

CASO DI STUDIO: I NUMERI VERTICALI

scrivere una funzione

```
void write_vertical(int n)
```

che prende un intero e lo stampa a video con tutte le cifre incolonnate (a partire da quella più significativa)

algoritmo:

- **caso più semplice:**
  se `n` è una sola cifra, allora scrivi il numero
- **caso tipico:**
  1) stampa il numero incolonnato meno l'ultima cifra
  2) scrivi l'ultima cifra

osservazione: il sottoproblema 1 è una versione più semplice del problema originale; il sottoproblema 2 è il caso più semplice

CASO DI STUDIO: I NUMERI VERTICALI

l'algoritmo `write_vertical` (**pseudo-codice**):

```
if (n < 10){
      cout << n << endl;
} else  // n ha due o più cifre
{
      write_vertical(n senza l'ultima cifra);
      cout << ultima cifra di n << endl;
}
```

CASO DI STUDIO: I NUMERI VERTICALI

la traduzione in **C++** dello pseudo-codice di **`write_vertical`**:

- `n/10` ritorna `n` con l'ultima cifra rimossa

  esempio: `124/10 = 12`

- `n%10` ritorna l'ultima cifra di `n`

  esempio: `124%10 = 4`

osservazione: rimuovere la cifra più significativa, stamparla, e applicare l'algoritmo alla restanti cifre sarebbe stato molto più complicato

CASO DI STUDIO: I NUMERI VERTICALI/IL CODICE

```
void write_vertical(int n) ;
    // Precondition: n >= 0
    // Postcondition: stampa le cifre di n incolonnate a partire
    // dalla più significativa

void write_vertical(int n) {
    if (n <= 9) cout << n << endl;
    else {   write_vertical (n/10) ;
             cout << n%10 << endl ;
    }
}
```

esempi di output:

```
write_vertical(123)  1
                     2
                     3
```

```
write_vertical(1)  1
```

ESEGUIRE UNA CHIAMATA RICORSIVA

```
OUTPUT WINDOW
> cout << 1 << endl ;
> cout << 12%10 << endl ;
> cout << 123%10 << endl ;
```

```
write_vertical(123)
 if (123 <= 9) cout << 123 << endl;
 else { write_vertical (123/10) ;
        cout << 123%10 << endl ;
 }
```

```
write_vertical(12)
 if (12 <= 9) cout << 12 << endl;
 else { write_vertical (12/10) ;
        cout << 12%10 << endl ;
 }
```

```
write_vertical(1)
 if (1 <= 9) cout << 1 << endl;
 else { write_vertical (1/10) ;
        cout << 1%10 << endl ;
 }
```

il caso
più semplice ora è
vero!

ANALIZZIAMO LA RICORSIONE

- la ricorsione in `write_vertical`
  - non necessita di nessuna parola chiave "**nuova**"
  - si riduce a invocare se stessa con argomenti differenti

- le chiamate ricorsive sono eseguite

1. **fermando temporaneamente l'esecuzione del chiamante** in corrispondenza della chiamata ricorsiva (il cui risultato è necessario per il chiamante)
2. **salvando le informazioni del chiamante per continuare l'esecuzione** in seguito (il program counter, per esempio)
3. **valutando la chiamata ricorsiva**
4. **ripristinando l'esecuzione del chiamante**

COME TERMINA LA RICORSIONE?

prima o poi una delle chiamate ricorsive non dipende da un'altra chiamata (abbiamo raggiunto il caso semplice)

formato dei problemi che hanno soluzione ricorsiva:

- ci sono uno o più casi (***casi induttivi***) il cui problema può essere risolto riconducendolo a problemi su casi più semplici
- ci sono uno o più casi semplici (***casi base o casi di fermata***) per i quali esiste una soluzione immediata, non ricorsiva

RICORSIONE "INFINITA"

una funzione ***che non raggiunge mai i casi di base***, *in teoria*, non termina mai

in realtà, spesso, il computer termina la memoria (ram) e il programma termina con eccezione "`out_of_memory`"

esempio: la funzione `write_vertical`, senza il caso di base

```
void new_write_vertical(int n){
      new_write_vertical (n /10);
      cout << n%10 << endl;
}
```

invocherà `new_write_vertical(0)`, che a sua volta invocherà `new_write_vertical(0)`, che a sua volta invocherà `new_write_vertical(0)`, che a sua volta invocherà `new_write_vertical(0)`. . . .

LA RICORSIONE E LE PILE

per eseguire una funzione ricorsiva i computer usano le pile

una pila è una struttura di memoria simile a una "**pila di carta**"

- per mettere le informazioni sulla pila, uno **scrive su un nuovo foglio di carta** e **lo mette sulla pila**
- per mettere altre informazioni sulla pila, uno prende ancora un nuovo foglio di carta, ci scrive sopra e lo mette sulla pila
- per **recuperare le informazioni dalla pila**, **è possibile leggerle solamente dal foglio di carta che è sulla pila**, che occorre buttare prima di leggere quelle del foglio sottostante

LAST-IN/FIRST-OUT (LIFO)

una pila è una struttura di memoria **last-in/first-out**

- l'ultimo elemento che è stato inserito è il primo ad essere rimosso

quando una funzione è invocata, il computer è come se usasse **un nuovo foglio di carta**

- la definizione della funzione è copiata su un foglio
- i parametri attuali rimpiazzano i parametri formali
- il computer inizia ad eseguire il corpo della funzione

PILA E CHIAMATE RICORSIVE

quando l'invocazione di una funzione esegue una chiamata ricorsiva:

- l'esecuzione del chiamante viene fermata (congelata) e l**e informazioni sono salvate sul foglio di carta corrente** (per consentire il ripristino dell'esecuzione più tardi)
- il **foglio di carta corrente è posto sulla pila**
- un **nuovo** foglio di carta è usato, in cui si copia la definizione della funzione e i suoi argomenti
- l'**esecuzione della funzione chiamata inizia**

PILA E RITORNO DI FUNZIONI RICORSIVE

quando **una chiamata ricorsiva termina** (non invoca altre funzioni)

- il computer butta via il foglio relativo alla funzione e recupera il foglio che si trova in testa alla pila
- continua l'esecuzione corrispondente (il chiamante) grazie alle informazioni memorizzate nel foglio
- quando l'esecuzione termina, si scarta il foglio e si prende quello successivo
- il processo continua finché non rimane alcun foglio

RECORD DI ATTIVAZIONE

il computer non usa fogli di carta o risme di carta ma usa porzioni di memoria (ram), dette

record di attivazione

- i record di attivazione **non contengono copie del codice** (occuperebbe molto spazio) ma soltanto l'indirizzo dell'istruzione nell'unica copia del codice in memoria

stack overflow: poiché ogni chiamata corrisponde a porre un nuovo record di attivazione in memoria, la **ricorsione infinita causa il consumo dello spazio riservato alla pila** (stack overflow)

- la computazione termina con errore `out of memory`

ESERCIZI

1. scrivere una funzione ricorsiva che prende **`n`** e stampa **`n`** asterischi
2. scrivere **`write_vertical`** in maniera che le cifre sono stampate dalla meno significativa alla più significativa
3. scrivere una funzione ricorsiva che prende **`n`** e stampa un numero di asterischi uguale alla somma dei quadrati dei primi **`n`** numeri naturali

RICORSIONE E ITERAZIONE

la ricorsione e l'iterazione sono due meccanismi computazionali equivalenti

- ogni problema che può essere risolto in maniera ricorsiva può anche essere risolto con cicli `while` o `for`
- una soluzione non ricorsiva di un problema è detta **soluzione iterativa**

osservazioni: la versione ricorsiva

- è meno efficiente di quella iterativa (calcola in più tempo)
- usa più memoria di quella iterativa
- l'algoritmo può essere *più elegante e compatto e deriva in modo naturale dalla definizione matematica*

FUNZIONI RICORSIVE CHE RITORNANO VALORI

le funzioni ricorsive possono anche tornare valori (come le altre funzioni)

la tecnica per definirle è uguale a quella già vista:

- ci sono uno o più casi in cui per ritornare un valore si invoca la funzione su argomenti "**più semplici**"
- ci sono uno o più casi in cui il valore è ritornato **senza dover invocare alcuna funzione**

ESEMPIO: IL FATTORIALE

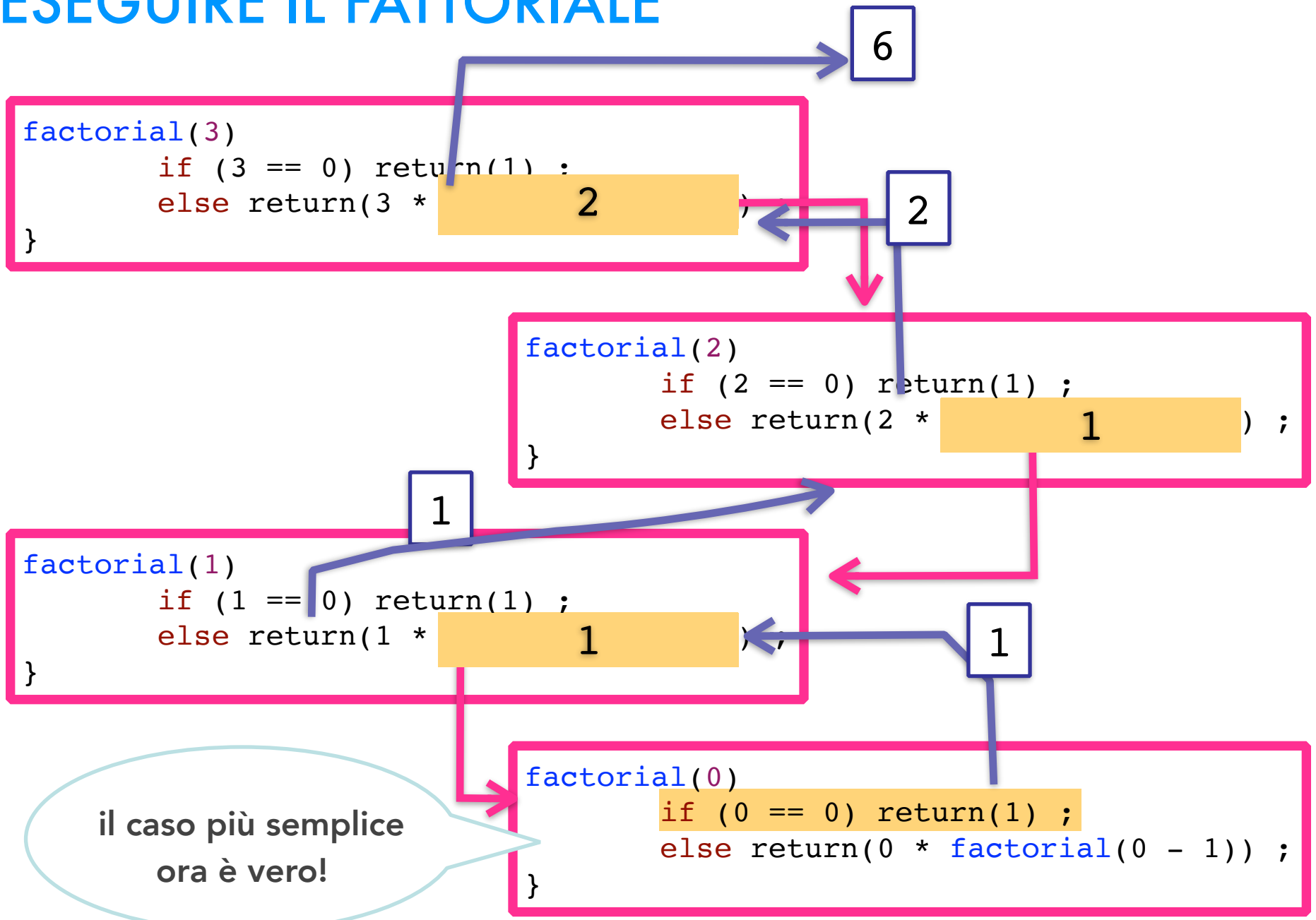
fattoriale

$$0! = 1$$
$$n! = n*(n-1)!$$

```
// Precondition:  n >= 0
// Postcondition: compute n! using a recursive definition
int factorial(int n){
      if (n == 0) return(1) ;
      else return(n * factorial(n - 1)) ;
}
```

osservazione: il programma è simile alla definizione matematica

ESEGUIRE IL FATTORIALE



ALTRO ESEMPIO: IL MCD CON L'ALGORITMO DI EUCLIDE

```
MCD(m, n)  = m                      se m=n
MCD(m, n)  = MCD(m-n, n)    se m>n
MCD(m, n)  = MCD(m, n-m)    se n>m
```

```
// Precondition: m,n are greater than 1
// Postcondition: calcola MCD(m, n) ricorsivamente
//                con l'algoritmo di Euclide
int MCD(int m, int n){
        if (m == n) return(m) ;
        else if (m > n) return(MCD(m-n, n)) ;
        else return(MCD(m, n-m));
}
```

LA RICORSIONE/FIBONACCI

numeri di Fibonacci

```
fib (0)  =          0
fib (1)  =          1
fib (n+2)= fib (n+1) + fib (n)
```

```
//  Precondition: n>= 0
//  Postcondition: calcola il numero n di Fibonacci

int fib (int n){
    if (n == 0) return(0) ;
    else if (n == 1) return(1);
    else return(fib(n - 1) + fib(n - 2));
}
```

osservazione: l'algoritmo è più complicato nella versione iterativa:

- *la versione ricorsiva è più elegante e compatta e deriva in modo naturale dalla definizione matematica*
- è **difficile individuare l'algoritmo iterativo** che implementa la versione ricorsiva

EFFICIENZA DI `FIB`

di solito la versione ricorsiva delle funzioni è meno efficiente della iterativa

esempio: in fibonacci i valori intermedi vengono ricalcolati ogni volta che occorrono

`fib(3)` viene calcolato due volte, `fib(2)` viene calcolato tre volte!

EFFICIENZA DI `FIB_IT`

```
// Precondition: n>=0
// Postcondition: calcola fibonacci di n in modo iterativo
int fib_it(int n){
        int a, b, c ;
        if (n == 0) return(1) ;
        else {    c = 1 ; b = 1 ; a = 0 ;
                  while (n>=1) {
                            c = b+ a;
                            a = b ;
                            b = c ;
                            n = n-1 ;
                  }
                  return(c) ;
        }
}
```

`fib_it` ha un numero lineare di assegnamenti a fronte di un numero esponenziale di `fib`

ESERCIZI

1. calcolare la seguente funzione

```
f(n) = n * n  +  f(n-2)
f(1) = 1
f(0) = 0
```

2. le operazioni sui naturali sono definite in modo ricorsivo
   - `n + 0 = n`
     `n + m = successore(n + (m-1))`
   - `n * 0 = 0`
     `n * m = (n * (m-1)) + n`

   scrivere un programma che calcola somma e prodotto in maniera ricorsiva

3. la ricorsione deve essere "ben-fondata". Cosa accade a

```
f(n) = f(n+1) +1
f(0) = 0
```

CASO DI STUDIO: INVERSIONE DI CARATTERI IN INPUT

problema: scrivere una funzione che prende in input una sequenza di caratteri che termina con il "." e la stampa in maniera invertita

- la lunghezza della sequenza è indefinita.

```
// Postcondition:   inverte una sequenza di caratteri che
//                  termina con "."
void revert(){
        char c ;
        cin >> c ;
        if (c != '.') {     revert() ;
                            cout << c ;
                    }
        else cout << "\n la sequenza invertita e`: " ;
}
```

problema: scriverlo mediante l'iterazione

REVERT ITERATIVA: IL PROGETTO DI REVERT_IT

revert funziona perchè utilizza la pila di record di attivazione

- se si vuole implementare `revert` in maniera iterativa, occorre definire una struttura pila
- in questo caso **la pila è una lista di elementi che contengono un carattere**

```
struct stack  {
          char val;
          stack *next;
} ;
```

```
void revert_it(){
    1. prende in input il primo carattere
    2. crea una lista di un solo elemento
    3. continua a prendere caratteri e a metterli in testa
       alla lista finchè non trova '.'
    4.  stampa la lista elemento per elemento
}
```

L'IMPLEMENTAZIONE DI `REVERT_IT`

```
void revert_it(){
        char c ;
        stack *p , *q;
        p = NULL ;    // p e` il puntatore alla testa della pila
        cin >> c ;
        while (c != '.') {
                q = new stack ; // q e` il puntatore al nuovo el.
                q->val = c ;
                q->next = p ;
                p = q ;
                cin >> c ;
        }
        cout << "\n la sequenza invertita e`: " ;
        while (p != NULL){ cout << p->val ; p = p->next ; }
    }
```

osservazione: non è possibile implementare `revert_it` senza strutture dati dinamiche!

MUTUA RICORSIONE

due o più funzioni sono *mutuamente ricorsive* quando una è definita in termini dell'altra

problema: definire in C++ le funzioni:

```
f(0) = 0                    g(0) = 1
f(n+1) = 1 + g(n)           g(n+1) = n * f(n)
```

il codice

```
int f (int n){
        if (n==0) return(0) ; else return(1+g(n-1)) ;
}
int g (int n){
        if (n==0) return(1) ; else return(n*f(n-1)) ;
}
```

è sbagliato! (*commento*: usare i prototipi di funzione)

ESERCIZI

1. scrivere un programma che prende in input una sequenza di cifre positive, a meno di uno 0 come elemento centrale e dice se la sequenza è palindroma oppure no. Si assuma che le lunghezze delle sequenze prima dello 0 e dopo lo 0 siano uguali

2. risolvere 1 quando le sequenze prima dello 0 e dopo lo 0 possono avere lunghezza differente e la sequenza termina con il "." (in questo caso la sequenza totale termina quando l'input è un valore diverso da 0, 1, …, 9)

COME PROGETTARE ALGORITMI RICORSIVI

quando si progetta una funzione ricorsiva (che ritorna un valore) **non occorre verificare tutte le possibili esecuzioni**

- **verifica che non ci siano ricorsioni infinite**: prima o poi si raggiunge sempre un caso di base
- **verifica che ogni caso di base ritorna il valore corretto**
- **assumendo** che le invocazioni ricorsive ritornino il valore corretto, **verificare che il valore ritornato dai casi con chiamata ricorsiva sia corretto**

osservazione: il progetto delle funzioni ricorsive che non ritornano valori è simile

CASO DI STUDIO: LA RICERCA BINARIA

specifica: cercare un valore `v` in un array `a` ordinato

- gli indici dell'array sono da `0` a `max_index`
- poichè l'array è ordinato, sappiamo che

```
a[0] <= a[1] <= . . .  <= a[max_index]
```

- se `v` si trova in `a`, vogliamo conoscerne la posizione
- la funzione prenderà in input il valore `v`, l'array `a` e il valore `max_index`
- la funzione ritornerà un intero: se l'intero è compreso tra `0` e `max_index` allora esso rappresenta l'indice di `a` dove è memorizzato `v`, se l'intero è `-1` significa che `v` non è stato trovato

LA RICERCA BINARIA: PRECONDIZIONI E POSTCONDIZIONI

```
// Precondition: il valori da a[0] a a[max_index]
// sono ordinati in maniera crescente

// Postcondition: se il valore v non si trova in
// a, allora la funzione ritorna -1, altrimenti
// ritorna l'indice i per cui a[i] = v
```

LA RICERCA BINARIA: ALGORITMO

l'algoritmo è il seguente:

1. cerca nel mezzo dell'array, cioè alla posizione `max_index/2`
2. se `a[max_index/2] == v` allora ritorna `max_index/2`
3. se `a[max_index/2] > v` allora bisogna cercare il valore `v` nella prima metà dell'array
4. se `a[max_index/2] < v` allora bisogna cercare il valore v nella seconda metà dell'array

RICERCA BINARIA: PRIMO PSEUDO-CODICE

```
mid = max_index/2 ;
if (a[mid] == v) return(mid) ;
else if (a[mid] > v) ricerca v da a[0] a a[mid-1] ;
else ricerca v da a[mid+1] a a[max_index] ;
```

osservazioni:

- l'algoritmo è **ricorsivo** (cercare su porzioni più piccole è più semplice che cercare sulla porzione più grande)
- per avere un algoritmo ricorsivo **occorre aumentare i parametri** della ricerca per specificare la porzione di array su cui si lavora
- bisogna introdurre un `min_index`

RICERCA BINARIA: SECONDO PSEUDO-CODICE

```
mid = (max_index + min_index) /2
if (a[mid] == v) return(mid)
else if (a[mid] > v)
          ricerca v da a[min_index] a a[mid-1]
else      ricerca v da a[mid+1] a a[max_index]
```

questioni:

- lo pseudo-codice **termina**?
  - **ogni invocazione ricorsiva si applica ad un array più corto**
- se il valore **v** non è presente nell'array, cosa succede?
  - ci sarà una chiamata ricorsiva in cui `min_index > max_index`

ALGORITMO DI RICERCA BINARIA: IL CODICE

```
int ricerca(int v, const int a[], int min_index, int max_index) ;

// Precondition: il valori da a[0] a a[max_index]
// sono ordinati in maniera crescente

// Postcondition: se il valore v non si trova in
// a, allora la funzione ritorna -1, altrimenti
// ritorna l'indice i per cui a[i] = v

int ricerca(int v, const int a[], int min_index, int max_index){
   if (min_index > max_index) return(-1) ;
   else { int mid = (max_index + min_index)/2 ;
        if (a[mid] == v) return(mid) ;
        else if (a[mid] > v)
                  return( ricerca(v,a,min_index,mid-1) ) ;
        else return( ricerca(v,a,mid+1, max_index) ) ;
       }
}
```

CODICE: TERMINAZIONE E CORRETTEZZA

la ricorsione **termina** perchè

- ad ogni chiamata ricorsiva la distanza tra `max_index` e `min_index` **decresce strettamente**
- la condizione di uscita è quando la distanza diventa **negativa** oppure quando viene trovato il valore `v`

il programma è **corretto** perchè

1. o l'array `a` "è inesistente", cioè

```
max_index - min_index < 0
```

2. oppure il valore `v` viene trovato (al centro)
3. oppure, quando `a[(max_index - min_index)/2] > v`, visto che `a` è ordinato, se `v` si trova in `a`, si troverà tra gli indici `(max_index-min_index)/2+1` e `max_index`
4. discorso simile quando `a[(max_index-min_index)/2]< v`

RICERCA BINARIA: EFFICIENZA

la ricerca binaria è estremamente efficiente rispetto a un algoritmo che ricerca un valore in un array facendo una scansione dell'array

- l'algoritmo di ricerca binaria **elimina metà elementi alla volta**
- su un array di 1000 elementi, la ricerca binaria esegue **al massimo 10 confronti** $= \log_2 1000$
- un algoritmo che fa una scansione sequenziale fa in media **500** confronti (il valore si trova a metà) e fa **1000** confronti nel caso pessimo (il valore non si trova)

RICERCA BINARIA: VERSIONE ITERATIVA

```
int ricerca(int v, const int a[], int min_index, int max_index) ;

// Precondition: il valori da a[0] a a[max_index]
// sono ordinati in maniera crescente

// Postcondition: se il valore v non si trova in
// a, allora la funzione ritorna -1, altrimenti
// ritorna l'indice i per cui a[i] = v

int ricerca(int v, const int a[], int min_index, int max_index){
      bool found = false ;
      int mid ;
      while ((min_index <= max_index) && !found){
            mid = (max_index + min_index)/2 ;
            if (a[mid] == v) found = true ;
            else if (a[mid] > v) max_index = mid - 1 ;
            else min_index = mid + 1 ;
      }
      if (found) return(mid) ; else return(-1) ;
}
```

ARRAY/MERGESORT

- è un algoritmo di ordinamento ricorsivo
- **casi base**: un array di 0 o 1 elemento è ordinato
- **caso induttivo**: dividere l'array in 2 metà, ordinarle tramite chiamate ricorsive e poi farne la fusione ordinata

```
void mergesort(int a[], int l, int r){
        int m;
        if (l<r) {
                m = (l+r)/2;
                mergesort(a, l, m);
                mergesort(a, m+1, r);
                merge(a, l, m, r);
        }
}
```

ARRAY/MERGESORT

```
void merge(int a[], int l, int m, int r){
        int i, j, k, b[length];
        i=l; k=l; j=m+1;
        while(i<=m && j<=r){
                if(a[i]<=a[j]){
                        b[k]=a[i];
                        i=i+1;
                } else {
                        b[k]=a[j];
                        j=j+1;
                }
                k=k+1;
        }
        if (i<=m){
           for(int h=m; h>=i; h=h-1) a[h+r-m]=a[h];
        }
        for(j=l; j<k; j=j+1) a[j]=b[j];
}
```

usa un secondo array!

copia due porzioni ordinate di array nel secondo array in modo da ottenere un array ordinato

copia della parte restante

ALGORITMI RICORSIVI SU LISTE

stampa degli elementi memorizzati in una lista:

```
void stampa_el_it(lista *p){                    // iterativa
    while (p != NULL) {
            cout << p->val ;
            p = p->next ;
    }
}

void stampa_el(lista *p){                       // ricorsiva
    if (p != NULL) {
            cout << p->val << eol;
            stampa_el(p->next) ;
    }
}
```

ALGORITMI RICORSIVI SU LISTE

ricerca di un elemento nella lista

```
bool search_el_it(lista *p, const int e){   // iterativa
     bool f = false ;
     while ((p != NULL) && (!f)){
          if (p->val == e) f = true ;
          else p = p->next ; }
     return(f) ;
}

bool search_el(lista *p, const int e){      // ricorsiva
     if (p == NULL) return(false) ;
     else if (p->val == e) return(true) ;
     else return(search_el(p->next, e)) ;
}
```

ESERCIZI

1. scrivere una funzione ricorsiva che prende una lista di interi e ritorna la somma degli interi nella lista

2. scrivere una funzione ricorsiva che prende una lista e ritorna la lista invertita

3. scrivere una funzione ricorsiva che prende una lista ed un intero e rimuove dalla lista la prima occorrenza dell'intero

4. scrivere una funzione ricorsiva che prende una lista ed un intero e rimuove dalla lista tutti i multipli dell'intero [confronta la tua soluzione con quella a pag. 4]