

Contents

Ricorsione	1
Modello di memoria della ricorsione	2
Ricorsione lineare	3
Progettare algoritmi ricorsivi	4
Il wrapper	4
Ricorsione su stringhe	5
Ricorsione su array	5
Ricorsione crescente	5
Ricorsione crescente con intervalli	6
Ricorsione decrescente con intervalli	6
Ricorsione dicotomica su array con intervalli	6
Ricorsione in lunghezza minima su coppie di array	7
Ricorsione in lunghezza massima su coppie di array	7
Scrivere funzioni ricorsive	8
Ricerca binaria(o dicotomica)	8
Ricorsione su matrici	9
Ricorsione non lineare	9
Serie di Fibonacci	9

Ricorsione

Funzioni definite in 2 passi:

- Caso base
- Caso generico

Prendiamo ad esempio la funzione fattoriale:

$$n! = \begin{cases} 1 & \text{se } n == 0 \\ n * (n-1)! & \text{altrimenti} \end{cases}$$

Questa funzione può essere definita iterativamente.

Tuttavia la struttura della funzione fattoriale che abbiamo sopra(caso base+generico) la rende nativamente compatibile con la dichiarazione ricorsiva di funzione

Se una funzione è ricorsiva possiamo avere solo 2 casi come visto sopra.

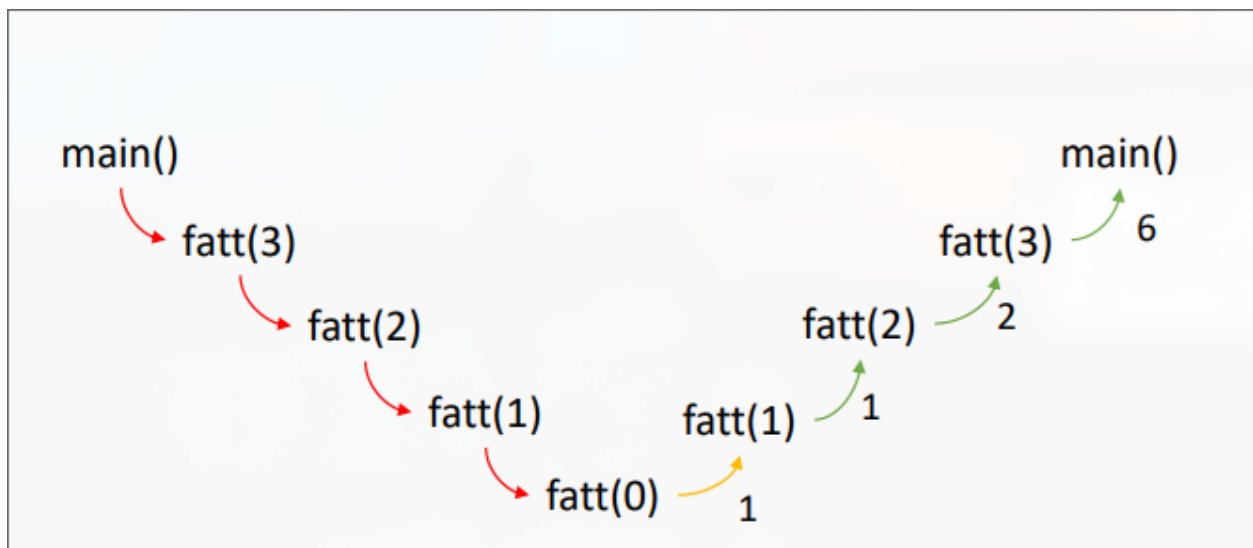
Dato il seguente codice per il calcolo del fattoriale:

- La funzione fattoriale è per definizione composta da un caso *base* e un caso *generale*

$$n! = \begin{cases} 1 & \text{se } n == 0 \\ n * (n-1)! & \text{altrimenti} \end{cases}$$

```
void main(void) {  
    int res = fatt (3);  
    printf ("3! vale %d\n", res);  
}  
  
int fatt(int n) {  
    int ret = 0;  
    if (n == 0) {  
        ret = 1;  
    }  
    else {  
        ret = n * fatt(n-1);  
    }  
    return ret;  
}
```

Abbiamo le seguenti chiamate di funzione:



Modello di memoria della ricorsione

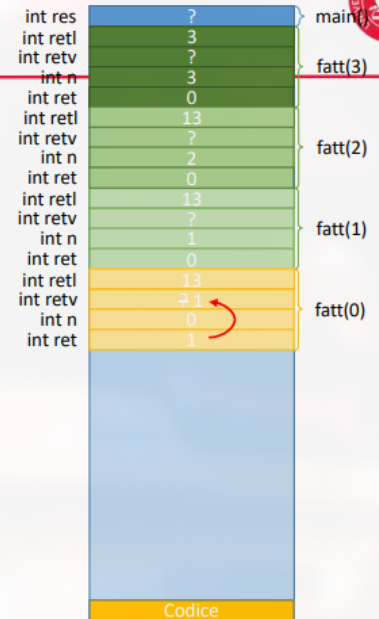
Ad ogni esecuzione della funzione viene creato un nuovo frame della funzione:

Modello di memoria

```
void main(void) {
    int res = fatt (3);
    printf ("3! vale %d\n", res);
}

int fatt(int n) {
    int ret = 0;
    if (n == 0) {
        ret = 1;
    }
    else {
        ret = n * fatt(n-1);
    }
    return ret;
}
```

Termina
l'esecuzione
di *fatt(0)*

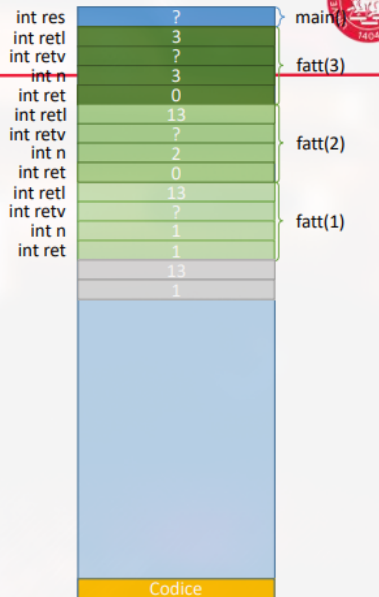


Quando le funzioni termineranno, il frame verrà cancellato.

Modello di memoria

```
void main(void) {
    int res = fatt (3);
    printf ("3! vale %d\n", res);
}

int fatt(int n) {
    int ret = 0;
    if (n == 0) {
        ret = 1;
    }
    else {
        ret = n * fatt(n-1);
    }
    return ret;
}
```



Attilio Fiandrotti, Programmazione I-A 2024-25

65

L'eccessivo numero di chiamate a funzioni può creare uno stack overflow perchè non c'è più spazio nello stack per creare frame di funzioni.

Ricorsione lineare

Una ricorsione dove il valore di n decresce alla chiamata ricorsiva, viene detta lineare.

Progettare algoritmi ricorsivi

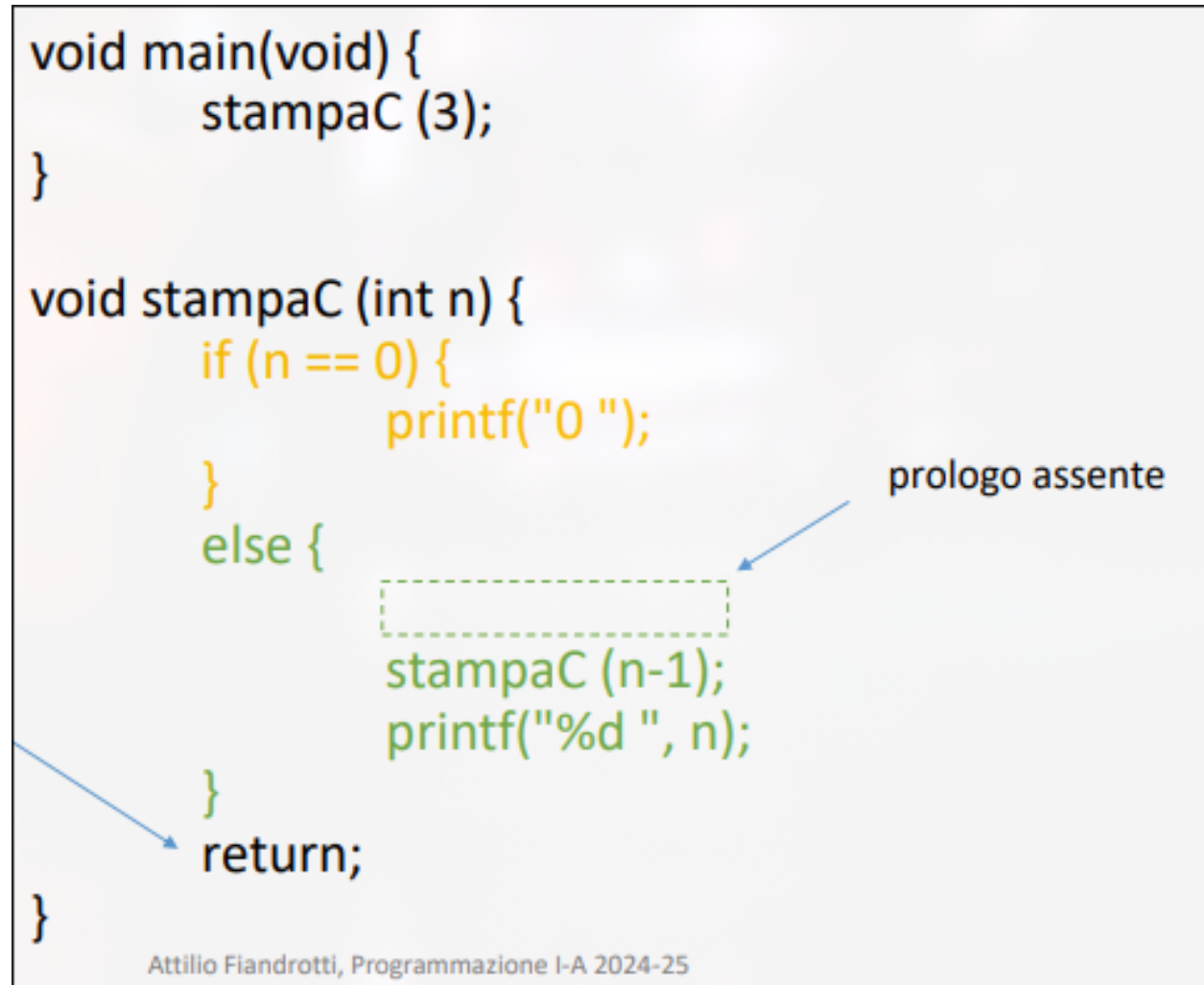
- Definiamo il prologo cioè il codice eseguito prima della chiamata ricorsiva
- Definiamo l'epilogo cioè il codice eseguito dopo la chiamata ricorsiva

Ad esempio questo è una funzione ricorsivo che stampa numeri in ordine crescente

```
void main(void) {
    stampaC (3);
}

void stampaC (int n) {
    if (n == 0) {
        printf("0 ");
    }
    else {
        stampaC (n-1);
        printf("%d ", n);
    }
    return;
}
```

prologo assente



Attilio Fiandrotti, Programmazione I-A 2024-25

In questo caso potremmo anche togliere il caso base

Possono esistere casi base composti da più condizioni.

Il wrapper

Il wrapper è una funzione che generalizza la chiamata a funzioni.

Ad esempio può prendere come input un solo parametro e ricavare gli altri a partire da quello.

Viene usato quando la funzione richiede due o più parametri che però possono essere ricavati da un parametro singolo.

Ricorsione su stringhe

Uguali all'array ma non ci serve la lunghezza della stringa perchè sappiamo che una stringa termina quando abbiamo '\0'

```
void stampaStringaR(const char a[], int n) {
    if (a[n] == '\0') {
    }
    else {
        stampaStringaR(a, n-1);
        printf("%c", a[n]);
    }
    return;
}
```

Il parametro n è il carattere di partenza della stringa. Il caso base è '\0' con le stringhe

Ricorsione su array

Esempio, stampa di un array da 0 a n crescente:

```
void stampaArrayC(int a[], int n) {
    if (n == 0) {
    }
    else {
        stampaArrayC(a, n-1);
        printf("%d", a[n-1]);
    }
    return;
}
```

Usiamo n-1 perchè stampiamo la lunghezza dell'array è l'elemento all'indice lunghezza array non esiste.

Per farlo in modo decrescente basta invertire la chiamata ricorsiva e il printf.

Ricorsione crescente

Finora abbiamo visto ricorsioni dove n decresceva.

Non c'entra con l'ordine di stampa/ordinamento

Viene detta anche **controvariante**

Il prototipo della funzione richiederà sia il minimo che il massimo.

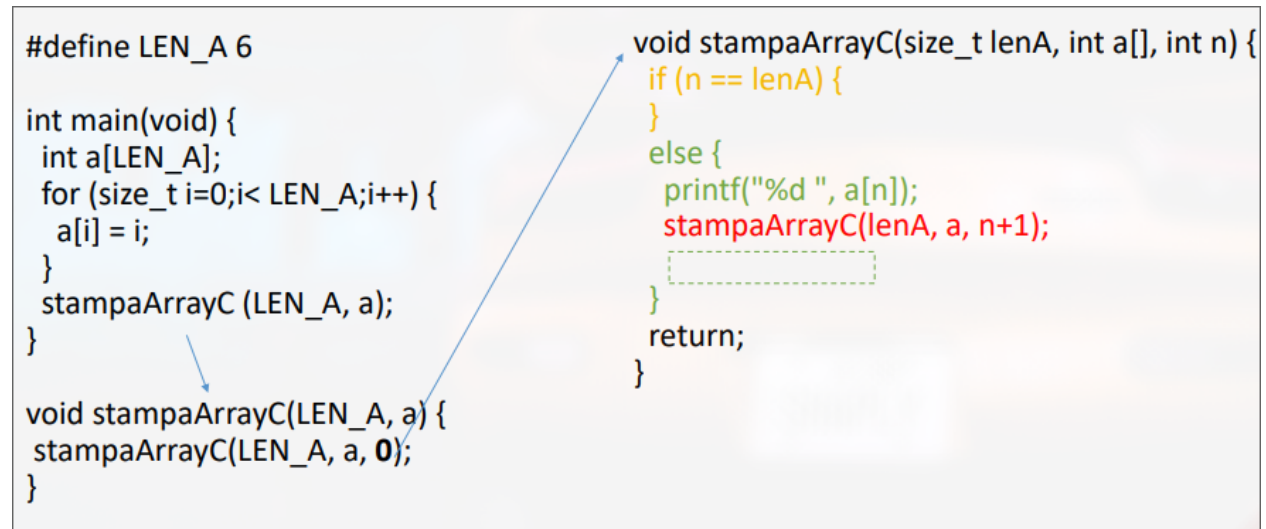
Il caso base sarà il massimo e non n == 0.

Ad esempio per stampare tutti i numeri in ordine crescente.

```
void stampaC(int n, int N) {
    if (n== N) {
    }
    else {
        printf("%d", n);
        stampaC (n+1, N);
    }
    return;
}
```

La ricorsione crescente è più intuitiva e presta meglio all'utilizzo con gli array

Inoltre si presta molto per la ricorsione su array.



Uso del wrapper

Ricorsione crescente con intervalli

Viene anche detta controvariante

```
if(left == right) { // Caso base
    return 0;
} else {
    return (a[left] - b[left]) + somma_diff_conR(a, b, left+1, right);
}
```

Ricorsione decrescente con intervalli

Viene anche detta covariante.

```
if(left == 0) {
    return 0;
} else {
    return a[left - 1] - b[left - 1]
        + somma_diff_covR(a, b, left - 1, right); // Elaborazioni con left-1 come indice
}
```

Ricorsione dicotomica su array con intervalli

Vengono visitati tutti gli elementi dell'array dividendo sempre l'array in due.

In generale:

```
int fnR(const int a[], const int b[],
        const size_t left, const size_t right) {
    if (left >= right) { // Oppure ==
        return 0; // caso base: intervallo vuoto, 0 è il valore base(in questo caso)
    }
    else if (right-left == 1) { // Cerco sempre un intervallo con almeno un elemento
```

```

        return a[left] - b[left]; // caso base: intervallo con 1 elemento. Eseguo l'algoritmo
    }
    else { // passo induttivo: dimezzamento intervallo
        int mid = (right + left) / 2;
        return fnR(a, b, left, mid) + fnR(a, b, mid, right);
    }
};

```

Ricorsione in lunghezza minima su coppie di array

Il wrapper calcola la lunghezza minima tra a e b e si passa alla funzione ricorsiva che itera fino alla lunghezza calcolata

Ricorsione in lunghezza massima su coppie di array

Dobbiamo controllare che l'array a o b sia definiti:

```

retType fnR(const size_t aLen, const int a[],
            const size_t bLen, const int b[], const size_t i)
{
    if (i >= aLen && i >= bLen)
        return valbase; // caso base
    else {
        if (i >= bLen) // a[i] esistente, b[] terminato
            return E(a[i], fnR(aLen, a, bLen, b, i+1));
        else if (i >= aLen) // b[i] esistente, a[] terminato
            return E(b[i], fnR(aLen, a, bLen, b, i+1));
        else // coppia a[i] e b[i] esistente
            return E(a[i], b[i], fnR(aLen, a, bLen, b, i+1));
    }
}

retType fn(const size_t aLen, const int a[],
            const size_t bLen, const int b[]) { // involucro
    return fnR(aLen, a, bLen, b, 0)
}

```

Per farlo con covariante basta seguire il metodo per scrivere funzioni su array con covariante (vengono cambiati gli indici in i-1 e il caso base i == 0. Vengono invertiti >= in <=), esempio:

```

retType fnR(const size_t aLen, const int a[],
            const size_t bLen, const int b[], const size_t i)
{
    if (i == 0)
        return valbase; // caso base
    else {
        if (i <= bLen) // a[i] esistente, b[] terminato
            return E(a[i], fnR(aLen, a, bLen, b, i+1));
        else if (i <= aLen) // b[i] esistente, a[] terminato
            return E(b[i], fnR(aLen, a, bLen, b, i+1));
        else // coppia a[i] e b[i] esistente
            return E(a[i], b[i], fnR(aLen, a, bLen, b, i+1));
    }
}

retType fn(const size_t aLen, const int a[],

```

```

const size_t bLen, const int b[]) { // involucro
    return fnR(aLen, a, bLen, b, aLen > bLen ? aLen : bLen);
}

```

Per farla in maniera dicotomica basta seguire il metodo per scrivere funzioni su array dicotomiche senza ulteriori cambiamenti (eccetto gli indici corretti).

Scrivere funzioni ricorsive

- Identifichiamo la più piccola operazione che lavora su un singolo termineranno

Ad esempio nel caso della sommatoria dobbiamo vedere tutto quello che avviene prima dell'ultimo termine.

Ricerca binaria (o dicotomica)

Ad ogni dimezzamento dell'array si considera solo uno dei due intervalli in funzione di un criterio di ricerca.

Divido a metà l'array e vado alla ricerca dell'elemento.

La ricerca binaria funziona solo se l'array è ordinato.

Passi dell'algoritmi:

1. Dato un array di lunghezza $[l=0, r]$ e key un valore da cercare
2. Se $l == r$, la chiave è assente
3. Calcoliamo la metà: $middle = (l+r)/2$
4. Verifichiamo se $a[middle] == key$. Nel caso fosse vero ritorniamo l'indice middle.
5. Confrontiamo key con $a[middle]$, se $key < a[middle]$, $r = m$
6. Iteriamo il processo dal punto 3.
7. Se $key > a[middle]$, $l = m+1$

Un esempio di algoritmo senza ricorsione è:

```

size_t binarySearch(const int a[], int key, size_t left, size_t right) {
    size_t ret = NOTFOUND;
    while (left < right && ret == NOTFOUND) {
        size_t middle = (left + right) / 2;
        if (a[middle] == key) {
            ret = middle;
        }
        else if (a[middle] > key) {
            right = middle;
        }
        else { // a[middle] < key
            left = middle + 1;
        }
    } // end while
    return ret;
}

```

Un esempio ricorsivo è:

```

bool ricerca_binaria(const int a[], const int val, size_t *pIndice,
                    const size_t left, const size_t right)
{
    if (left > right) {
        return false; // Caso base: intervallo vuoto
    } else {
        int mid = (left + right) / 2;

```



```

    if (a[mid] == val) {
        *pIndice = mid; // L'elemento è stato trovato, aggiorniamo l'indice
        return true;
    } else {
        if (a[mid] > val) {
            return ricerca_binaria(a, val, pIndice, left, mid - 1); // Discesa a sinistra
        }
        else {
            return ricerca_binaria(a, val, pIndice, mid + 1, right); // Discesa a destra
        }
    }
}
}
}

```

Ricorsione su matrici

Usiamo due funzioni ricorsive, una che opera su colonne e una che opera sulle righe. Useremo le stesse proprietà che usiamo per gli array

Ricorsione non lineare

Serie di Fibonacci

La serie di Fibonacci inizia con 0 e 1 e ha come proprietà che ogni numero successivo è la somma dei due precedenti

La funzione sarà definita come:

- $\text{fibonacci}(0) = 0$
- $\text{fibonacci}(1) = 1$
- $\text{fibonacci}(n) = \text{fibonacci}(n-1) + \text{fibonacci}(n-2)$

Il caso base sarà definito come OR.

Come possiamo vedere, il caso generale è definito da due chiamate ricorsive, per questo non è più una ricorsione lineare.

Tutta la sequenza di ricorsioni generate verrà chiamato **albero di ricorsione**.