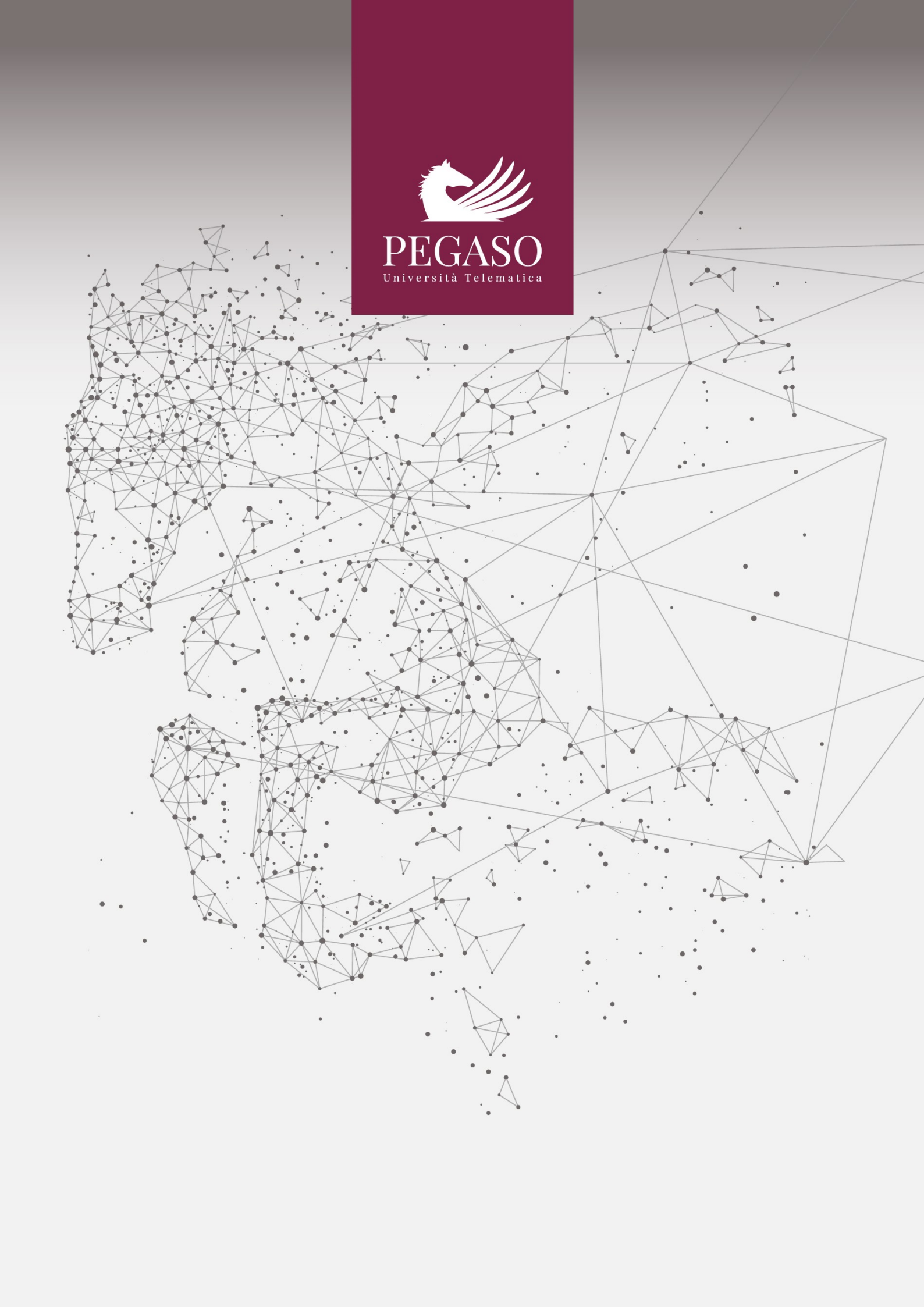

PEGASO
Università Telematica

Indice

1. RICORSIONE 3
2. CALCOLO RICORSIVO DEL FATTORIALE 5
3. LA RICORSIONE RISPETTO ALL'ITERAZIONE 11
4. PRESTAZIONI 13
RIFERIMENTI BIBLIOGRAFICI 14

1. Ricorsione

Per alcuni tipi di problemi è utile avere funzioni che chiamano se stesse.

Una funzione ricorsiva è una funzione che chiama se stessa direttamente, o indirettamente attraverso un'altra funzione.

Una funzione ricorsiva è chiamata per risolvere un problema.

La funzione "sa" in realtà soltanto come risolvere i casi più semplici, cioè i cosiddetti casi di base.

Se la funzione è chiamata con un caso di base, la funzione restituisce semplicemente un risultato.

Se la funzione è chiamata con un problema più complesso, solitamente divide il problema in due parti concettuali: una parte che sa come risolvere e una parte che non sa come risolvere direttamente.

Per rendere la ricorsione fattibile, quest'ultima parte deve somigliare al problema originario ma essere una versione leggermente più semplice o più piccola.

Poiché questo nuovo problema somiglia al problema originario, la funzione lancia

(chiama) una nuova copia di se stessa per lavorare sul problema più semplice.

Questo passo è detto chiamata ricorsiva o passo di ricorsione.

Il passo di ricorsione include anche un'istruzione return, perché il suo risultato sarà combinato con la porzione del problema che la funzione sa come risolvere per formare un risultato che sarà restituito alla funzione chiamante originaria.

Il passo di ricorsione viene eseguito mentre la chiamata originaria alla funzione si arresta,

in attesa del risultato dal passo di ricorsione.

Il passo di ricorsione può produrne molte di più di tali chiamate ricorsive, mentre la funzione continua a dividere in due parti concettuali ogni problema per il quale è chiamata.

Perché termini la ricorsione, ogni volta che la funzione chiama se stessa con una versione leggermente più semplice del problema originario, questa sequenza di problemi più semplici deve alla fine convergere al caso di base.

Quando la funzione riconosce il caso di base, restituisce un risultato alla copia precedente della funzione, e da qui segue una sequenza di restituzioni in cui si ripercorre all'indietro tutta la sequenza delle chiamate fino a che la chiamata originaria della funzione non restituisce infine il risultato finale alla sua funzione chiamante.

Come esempio di questi concetti all'opera, scriviamo un programma ricorsivo per eseguire un comune calcolo matematico.

2. Calcolo ricorsivo del fattoriale

Il fattoriale di un intero non negativo n, scritto n! (pronunciato "n fattoriale"), è il prodotto

$n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$

con 1! uguale a 1, e 0! definito come 1.

Ad esempio, 5! è il prodotto 5 * 4 * 3 * 2 * 1, che è uguale a 120.

Il fattoriale di un intero, `number`, maggiore o uguale a 0 si può calcolare iterativamente (non ricorsivamente) usando un'istruzione `for` come segue:

```
factorial = 1;
for (counter = number; counter >= 1; --counter)
    factorial *= counter;
```

Si arriva a una definizione ricorsiva della funzione fattoriale osservando la seguente relazione:

$n! = n \cdot (n-1)!$

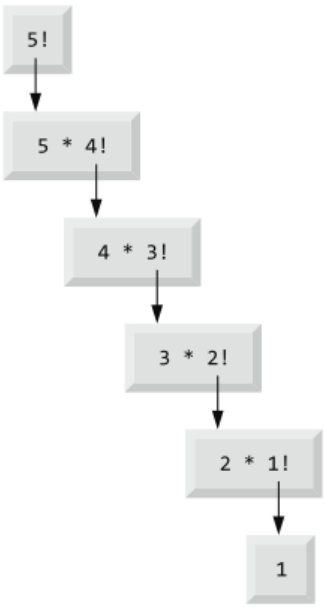
Ad esempio, 5! è chiaramente uguale a 5 * 4! come mostrato dalle seguenti espressioni:

$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$

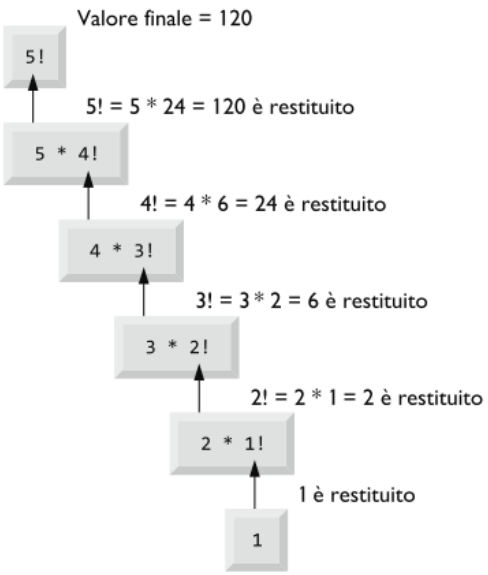
$5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1)$

$5! = 5 \cdot (4!)$

Il calcolo di 5! procede come mostrato nella seguente figura, che evidenzia come la successione di chiamate ricorsive proceda fino a che 1! è calcolato come 1 (cioè il caso di base), il che termina la ricorsione.



La Figura seguente mostra il valore restituito da ogni chiamata ricorsiva alla sua funzione chiamante, fino a che è calcolato e restituito il valore finale.



Il seguente programma usa la ricorsione per calcolare e stampare i fattoriali degli interi da 0 a 10:

```
1  // Programma in C per
2  // Funzione fattoriale ricorsiva.
3  #include <stdio.h>
4
5  unsigned long long int factorial(unsigned int number);
6
7  int main(void)
8  {
9     // durante ogni iterazione, calcola
10    // factorial(i) e stampa il risultato
11    for (unsigned int i = 0; i <= 21; ++i) {
12       printf("%u! = %llu\n", i, factorial(i));
13    }
14 }
15
16 // definizione ricorsiva della funzione fattoriale
17 unsigned long long int factorial(unsigned int number)
18 {
19    // caso di base
20    if (number <= 1) {
21       return 1;
22    }
23    else { // passo ricorsivo
24       return (number * factorial(number - 1));
25    }
26 }
```

```
0! = 1

1! = 1
```

```
2! = 2

3! = 6

4! = 24

5! = 120

6! = 720

7! = 5040

8! = 40320

9! = 362880

10! = 3628800

11! = 39916800

12! = 479001600

13! = 6227020800

14! = 87178291200

15! = 1307674368000

16! = 20922789888000

17! = 355687428096000

18! = 6402373705728000

19! = 121645100408832000

20! = 2432902008176640000

21! = 14197454024290336768
```

La funzione ricorsiva `factorial` verifica dapprima se una condizione di terminazione è vera, cioè se `number` è minore o uguale a 1.

Se `number` è davvero minore o uguale a 1, `factorial` restituisce 1; non è necessaria alcuna ulteriore ricorsione e il programma termina.

Se `number` è maggiore di 1, l'istruzione

```
return number * factorial(number - 1);
```

esprime il problema come il prodotto di `number` e di una chiamata ricorsiva a `factorial` che calcola il fattoriale di `number - 1`.

La chiamata `factorial(number - 1)` è un problema un po' più semplice dell'originario calcolo `factorial(number)`.

La funzione `factorial` (righe 17–26) riceve un `unsigned int` e restituisce un risultato di tipo `unsigned long long int`.

Il C standard specifica che una variabile di tipo `unsigned long long int` può assumere un valore almeno tanto grande quanto 18.446.744.073.709.551.615.

Come si può vedere dall'output del programma, i valori fattoriali diventano grandi velocemente.

Abbiamo scelto il tipo di dati `unsigned long long int` in modo che il programma possa calcolare valori fattoriali più grandi.

Lo specificatore di conversione `%llu` si usa par stampare valori `unsigned long long int`.

Purtroppo, la funzione fattoriale produce valori grandi così rapidamente che perfino `unsigned long long int` non ci permette di stampare molti valori fattoriali prima che sia superato il valore massimo di una variabile `unsigned long long int`.

Anche quando usiamo `unsigned long long int` non riusciamo ancora a calcolare i fattoriali oltre 21!

Questo mette in evidenza una debolezza del C (e di altri linguaggi di programmazione procedurali), nel senso che il linguaggio non può essere facilmente esteso per trattare requisiti specifici di varie applicazioni.

I linguaggi orientati agli oggetti, come il C++, sono in genere linguaggi estensibili che, con il meccanismo delle "classi", ci permettono la creazione di nuovi tipi di dati, compresi quelli che possono rappresentare interi arbitrariamente grandi se lo vogliamo.

☹ *Errore comune di programmazione*

Dimenticare di far restituire un valore a una funzione ricorsiva quando deve restituirne uno.

☹ *Errore comune di programmazione*

Omettere il caso di base, o scrivere il passo di ricorsione in modo scorretto così che esso non converga al caso di base, provocherà una ricorsione infinita, facendo esaurire alla fine la memoria.

Ciò è analogo al problema di un ciclo infinito in una soluzione iterativa (non ricorsiva).

3. La ricorsione rispetto all'iterazione

Il calcolo del fattoriale si può facilmente implementare o ricorsivamente o iterativamente. Confrontiamo i due approcci ed esaminiamo perché potreste sceglierne uno rispetto all'altro in una particolare situazione.

- Sia l'iterazione che la ricorsione si basano su una struttura di controllo: l'iterazione usa una struttura di iterazione; la ricorsione usa una struttura di selezione.
- Sia l'iterazione che la ricorsione implicano la ripetizione: l'iterazione usa esplicitamente un'istruzione di iterazione; la ricorsione attua la ripetizione attraverso ripetute chiamate di funzione.
- L'iterazione e la ricorsione richiedono ciascuna un test di terminazione: l'iterazione termina quando la condizione di continuazione del ciclo fallisce; la ricorsione quando si incontra un caso di base.
- L'iterazione con la iterazione controllata da contatore e la ricorsione procedono ciascuna gradualmente verso la terminazione: l'iterazione continua a modificare un contatore finché questo assume un valore che fa fallire la condizione di continuazione del ciclo; la ricorsione continua a produrre versioni più semplici del problema originario finché non viene raggiunto il caso di base.
- Sia l'iterazione che la ricorsione possono andare avanti all'infinito: si ha un ciclo infinito con l'iterazione se il test di continuazione del ciclo non diventa mai falso; si ha una ricorsione infinita se il passo di ricorsione non riduce ogni volta il problema in modo che esso converga al caso di base. L'iterazione e la ricorsione infinita solitamente si verificano come risultato di errori nella logica di un programma.

  La ricorsione ha molti difetti.

Essa invoca ripetutamente il meccanismo di chiamata di funzione, con conseguente aggravio di calcolo (detto overhead).

Ciò può essere dispendioso sia in termini di tempo di elaborazione sia in termini di spazio di memoria.

Ogni chiamata ricorsiva fa sì che sia creata un'altra copia della funzione (in realtà solo le variabili della funzione vengono copiate); questo può consumare una considerevole quantità di memoria.

L'iterazione è eseguita normalmente entro una funzione, così non si ha l'aggravio di calcolo delle ripetute chiamate di funzione e dell'assegnazione extra di memoria.

Allora, perché scegliere la ricorsione?

Osservazione di ingegneria del software

Qualunque problema che si può risolvere in maniera ricorsiva si può anche risolvere in maniera iterativa (non ricorsivamente).

Un approccio ricorsivo si preferisce normalmente a un approccio iterativo quando rispecchia in maniera più naturale il problema e produce un programma più facile da capire e da correggere.

Un'altra ragione per scegliere una soluzione ricorsiva sta nel fatto che una soluzione iterativa potrebbe non essere evidente.

4. Prestazioni

Funzionalizzare i programmi favorisce una buona ingegneria del software, ma ha un prezzo.

Un programma pesantemente funzionalizzato – confrontato con un programma monolitico (cioè di un solo modulo) senza funzioni – effettua potenzialmente un gran numero di chiamate di funzioni, e ciò consuma tempo d'esecuzione sul processore di un computer.

Benché i programmi monolitici possano avere prestazioni migliori, sono più difficili da scrivere, testare, correggere, mantenere e sviluppare.

Riferimenti bibliografici

- Paul Deitel, Harvey Deitel, "Il linguaggio C – Fondamenti e tecniche di programmazione", Libro edito da Pearson Italia. Include anche utili esercizi di autovalutazione.