Algoritmi in Linguaggio C

80 esercizi di programmazione e soluzioni commentate

Federico Bolelli e Maurizio Vincini

Indice

Pagina

1 Introduzione **7**
1.1 Com'è strutturato il libro e come va utilizzato? . . . 8
1.2 Scelte di Stile . . . 9
1.2.1 *Google C++ Style Guide* . . . 9
1.2.2 Scelte *Filosofiche* . . . 12
1.2.3 Scelte Pratiche e Dettagli sullo Standard del C . . . 12
1.2.4 Programmazione Strutturata . . . 13
1.3 File di Supporto e Funzioni Primitive . . . 14

2 Ricorsione **15**
2.1 Un Breve Ripasso . . . 15
2.1.1 Caso Base della Ricorsione . . . 16
2.1.2 Validazione dell'Input . . . 17
2.1.3 Ricorsione Diretta e Indiretta . . . 18
2.1.4 Ricorsione di Coda . . . 18
2.2 Esercizi . . . 20
2.2.1 Esercizio – fattoriale . . . 20
2.2.2 Esercizio – binomiale . . . 20
2.2.3 Esercizio – fibonacci . . . 21
2.2.4 Esercizio – sin-x . . . 21
2.2.5 Esercizio – newton . . . 21
2.2.6 Esercizio – secanti . . . 22
2.2.7 Esercizio – trailing . . . 23
2.2.8 Esercizio – palindromo . . . 23
2.2.9 Esercizio – hailstone . . . 24
2.2.10 Esercizio – inverti-array . . . 24
2.2.11 Esercizio – atoi . . . 24
2.2.12 Esercizio – power-two . . . 25
2.2.13 Esercizio – quoziente . . . 25
2.2.14 Esercizio – inverti-cifre . . . 26
2.2.15 Esercizio – MCD . . . 26
2.2.16 Esercizio – recaman . . . 26
2.2.17 Esercizio – frazione-egizia . . . 27

3 Backtracking **29**
3.1 Algoritmi di *Backtracking* . . . 29
3.2 Esercizi . . . 32

3.2.1 Esercizio – stringhe 32
3.2.2 Esercizio – stringhe-mod 33
3.2.3 Esercizio – subset-equal 33
3.2.4 Esercizio – regole 33
3.2.5 Esercizio – gruppi 34
3.2.6 Esercizio – gruppi-norep 35
3.2.7 Esercizio – n-steps 36
3.2.8 Esercizio – articoli 36
3.2.9 Esercizio – colormap 37
3.2.10 Esercizio – cacciatore-preda 38
3.2.11 Esercizio – gola-cresta 39
3.2.12 Esercizio – alpine-road 40
3.2.13 Esercizio – robot-grid 41
3.2.14 Esercizio – robot-grid-mod 43
3.2.15 Esercizio – queens 43
3.2.16 Esercizio – nonograms 45

4 Liste **47**
4.1 Introduzione alla Struttura Dati 47
4.2 Implementazione 48
4.2.1 `ElemType` 48
4.2.2 Primitive 52
4.3 Esercizi 57
4.3.1 Esercizio – length 57
4.3.2 Esercizio – occorrenze 58
4.3.3 Esercizio – coords-load 58
4.3.4 Esercizio – vector-list 58
4.3.5 Esercizio – taglia 59
4.3.6 Esercizio – split 59
4.3.7 Esercizio – concatena 60
4.3.8 Esercizio – concatena-n 60
4.3.9 Esercizio – filter 60
4.3.10 Esercizio – any-loop 61
4.3.11 Esercizio – vette 61
4.3.12 Esercizio – bucket-sort 61
4.3.13 Esercizio – common-tail 63
4.3.14 Esercizio – back2front 63
4.3.15 Esercizio – insert-ord 63
4.3.16 Esercizio – sum 64
4.3.17 Esercizio – insertion-sort 64
4.3.18 Esercizio – ordered-merge 65
4.3.19 Esercizio – remove-dup 65

5 Alberi **67**
5.1 Introduzione alla Struttura Dati 67
5.1.1 Definizioni e Proprietà di Alberi e Nodi 68
5.1.2 Alberi BST 70
5.1.3 Implementazione 70
5.1.4 Algoritmi di Visita 73

5.2 Esercizi . . . 75
5.2.1 Esercizio – preorder-delete . . . 75
5.2.2 Esercizio – mirror . . . 75
5.2.3 Esercizio – compare . . . 76
5.2.4 Esercizio – height . . . 76
5.2.5 Esercizio – read . . . 76
5.2.6 Esercizio – write . . . 78
5.2.7 Esercizio – vector2bst . . . 78
5.2.8 Esercizio – tree2bst . . . 78
5.2.9 Esercizio – avl . . . 79
5.2.10 Esercizio – vector2tree . . . 80
5.2.11 Esercizio – count . . . 81
5.2.12 Esercizio – prune . . . 81
5.2.13 Esercizio – mediano . . . 82
5.2.14 Esercizio – morris . . . 82
5.2.15 Esercizio – pred-succ . . . 83
5.2.16 Esercizio – level-order . . . 84
5.2.17 Esercizio – isomorfi . . . 85
5.2.18 Esercizio – equivalenti . . . 85
5.2.19 Esercizio – diameter . . . 86

6 Heap **87**
6.1 Introduzione alla Struttura Dati . . . 87
6.1.1 Implementazione . . . 88
6.2 Esercizi . . . 95
6.2.1 Esercizio – heapify . . . 95
6.2.2 Esercizio – remove . . . 95
6.2.3 Esercizio – copy . . . 95
6.2.4 Esercizio – is-heap . . . 95
6.2.5 Esercizio – get-pop . . . 96
6.2.6 Esercizio – heapsort . . . 96
6.2.7 Esercizio – kth-least . . . 96
6.2.8 Esercizio – merge . . . 96
6.2.9 Esercizio – min2max . . . 97

7 Soluzioni **99**
Soluzione dell'esercizio 2.2.1 – fattoriale . . . 99
Soluzione dell'esercizio 2.2.2 – binomiale . . . 101
Soluzione dell'esercizio 2.2.3 – fibonacci . . . 102
Soluzione dell'esercizio 2.2.4 – sin-x . . . 106
Soluzione dell'esercizio 2.2.5 – newton . . . 108
Soluzione dell'esercizio 2.2.6 – secanti . . . 112
Soluzione dell'esercizio 2.2.7 – trailing . . . 115
Soluzione dell'esercizio 2.2.8 – palindromo . . . 118
Soluzione dell'esercizio 2.2.9 – hailstone . . . 120
Soluzione dell'esercizio 2.2.10 – inverti-array . . . 122
Soluzione dell'esercizio 2.2.11 – atoi . . . 124
Soluzione dell'esercizio 2.2.12 – power-two . . . 130
Soluzione dell'esercizio 2.2.13 – quoziente . . . 131

Soluzione dell'esercizio 2.2.14 – inverti-cifre 132
Soluzione dell'esercizio 2.2.15 – MCD 134
Soluzione dell'esercizio 2.2.16 – recaman 135
Soluzione dell'esercizio 2.2.17 – frazione-egizia 138
Soluzione dell'esercizio 3.2.1 – stringhe 139
Soluzione dell'esercizio 3.2.2 – stringhe-mod 142
Soluzione dell'esercizio 3.2.3 – subset-equal 144
Soluzione dell'esercizio 3.2.4 – regole 150
Soluzione dell'esercizio 3.2.5 – gruppi 153
Soluzione dell'esercizio 3.2.6 – gruppi-norep 158
Soluzione dell'esercizio 3.2.7 – n-steps 161
Soluzione dell'esercizio 3.2.8 – articoli 163
Soluzione dell'esercizio 3.2.9 – colormap 170
Soluzione dell'esercizio 3.2.10 – cacciatore-preda 175
Soluzione dell'esercizio 3.2.11 – gola-cresta 181
Soluzione dell'esercizio 3.2.12 – alpine-road 184
Soluzione dell'esercizio 3.2.13 – robot-grid 188
Soluzione dell'esercizio 3.2.14 – robot-grid-mod 192
Soluzione dell'esercizio 3.2.15 – queens 198
Soluzione dell'esercizio 3.2.16 – nonograms 204
Soluzione dell'esercizio 4.3.1 – length 216
Soluzione dell'esercizio 4.3.2 – occorrenze 218
Soluzione dell'esercizio 4.3.3 – coords-load 221
Soluzione dell'esercizio 4.3.4 – vector-list 223
Soluzione dell'esercizio 4.3.5 – taglia 226
Soluzione dell'esercizio 4.3.6 – split 228
Soluzione dell'esercizio 4.3.7 – concatena 231
Soluzione dell'esercizio 4.3.8 – concatena-n 234
Soluzione dell'esercizio 4.3.9 – filter 236
Soluzione dell'esercizio 4.3.10 – any-loop 239
Soluzione dell'esercizio 4.3.11 – vette 243
Soluzione dell'esercizio 4.3.12 – bucket-sort 245
Soluzione dell'esercizio 4.3.13 – common-tail 248
Soluzione dell'esercizio 4.3.14 – back2front 251
Soluzione dell'esercizio 4.3.15 – insert-ord 253
Soluzione dell'esercizio 4.3.16 – sum 256
Soluzione dell'esercizio 4.3.17 – insertion-sort 261
Soluzione dell'esercizio 4.3.18 – ordered-merge 263
Soluzione dell'esercizio 4.3.19 – remove-dup 269
Soluzione dell'esercizio 5.2.1 – preorder-delete 273
Soluzione dell'esercizio 5.2.2 – mirror 277
Soluzione dell'esercizio 5.2.3 – compare 279
Soluzione dell'esercizio 5.2.4 – height 281
Soluzione dell'esercizio 5.2.5 – read 284
Soluzione dell'esercizio 5.2.6 – write 286
Soluzione dell'esercizio 5.2.7 – vector2bst 288
Soluzione dell'esercizio 5.2.8 – tree2bst 291
Soluzione dell'esercizio 5.2.9 – avl 293

Soluzione dell'esercizio 5.2.10 – vector2tree 296
Soluzione dell'esercizio 5.2.11 – count 299
Soluzione dell'esercizio 5.2.12 – prune 301
Soluzione dell'esercizio 5.2.13 – mediano 303
Soluzione dell'esercizio 5.2.14 – morris 306
Soluzione dell'esercizio 5.2.15 – pred-succ 309
Soluzione dell'esercizio 5.2.16 – level-order 317
Soluzione dell'esercizio 5.2.17 – isomorfi 323
Soluzione dell'esercizio 5.2.18 – equivalenti 325
Soluzione dell'esercizio 5.2.19 – diameter 327
Soluzione dell'esercizio 6.2.1 – heapify 329
Soluzione dell'esercizio 6.2.2 – remove 332
Soluzione dell'esercizio 6.2.3 – copy 335
Soluzione dell'esercizio 6.2.4 – is-heap 337
Soluzione dell'esercizio 6.2.5 – get-pop 340
Soluzione dell'esercizio 6.2.6 – heapsort 342
Soluzione dell'esercizio 6.2.7 – kth-least 344
Soluzione dell'esercizio 6.2.8 – merge 348
Soluzione dell'esercizio 6.2.9 – min2max 352

8 Indice degli Esercizi per Tipologia **355**
8.1 Ricorsione 355
8.2 Backtracking 356
8.3 Liste 357
8.4 Alberi 357
8.5 BST 358
8.6 Heap 358
8.7 Input/Output con File 358
8.8 Algoritmi di Ordinamento 358
8.9 Colloquio di Lavoro - Interview 359
8.10 Elementi di Teoria 359

Appendici **359**

A Algoritmo di Floyd - *La Lepre e la Tartaruga* **361**

Capitolo 1

Introduzione

Questo libro nasce dall'esperienza maturata dagli autori durante l'insegnamento di Fondamenti di Informatica I e II del corso di Laurea Triennale in Ingegneria Informatica dell'Università degli Studi di Modena e Reggio Emilia e vuole essere una guida per gli studenti alla preparazione e verifica delle proprie competenze.

Il libro colleziona 80 esercizi su algoritmi e strutture dati in linguaggio C. Oltre a fornire agli studenti uno strumento strutturato per la preparazione all'esame, il testo si pone l'obiettivo di stimolare l'apprendimento fornendo al lettore spunti di analisi e riflessione per andare oltre la semplice risoluzione dell'esercizio e comprendere appieno il passaggio dalla conoscenza astratta dei concetti all'implementazione pratica.

Ciascun capitolo è relativo ad una particolare tecnica di programmazione o a una struttura dati: algoritmi ricorsivi e risoluzione di problemi mediante la ricorsione, backtracking, liste, alberi e heap. Inoltre, tematiche fondamentali quali lo studio dell'analisi computazionale e gli algoritmi di ordinamento vengono affrontate in modo trasversale nella trattazione degli argomenti dei vari capitoli. In particolare, riguardo all'analisi computazionale si è scelto di non fornire quasi mai la dimostrazione matematica, quanto considerazioni specifiche e rigorose riguardo i costi in termini di tempo e memoria delle implementazioni proposte.

In generale, non esiste *la soluzione* ad un problema, ma esistono *le soluzioni*. Proprio per questo motivo, per molti degli esercizi vengono proposte due o più soluzioni alternative, evidenziando di volta in volta quali sono i vantaggi di una rispetta all'altra e, quando opportuno, confrontando il costo computazionale in termini di tempo e memoria delle alternative.

Infine, il Capitolo 8 riporta l'indice degli esercizi per argomento, trasversalmente alla suddivisione che abbiamo scelto per i capitoli precedenti. In questo modo il lettore potrà facilmente individuare problemi risolti con tecniche specifiche, come ad esempio l'ordinamento, o appartenenti a determinate categorie.

Essendo questo un libro di esercizi su algoritmi e strutture dati, in generale le nozioni fondamentali del C verranno date spesso per scontate: non mancano comunque casi nei quali abbiamo ritenuto opportuna e doverosa una spiegazione o chiarimento più dettagliato sull'uso del linguaggio.

1.1 Com'è strutturato il libro e come va utilizzato?

Ciascun capitolo è introdotto da una trattazione teorica che descrive la classe di algoritmi o la struttura dati astratta su cui si focalizzeranno gli esercizi presenti. In questa fase introduttiva vengono descritti gli aspetti fondamentali per la risoluzione dei problemi e, quando necessario, illustrate le *funzioni primitive* poi adottate nelle soluzioni. L'obiettivo è quello di fornire i concetti di base utili a comprendere e a risolvere gli esercizi, senza pretesa di esaustività.

Per funzioni primitive s'intende l'insieme minimo di funzioni o procedure che definiscono le operazioni di base necessarie per poter gestire una determinata struttura dati, ossia per poterla creare (allocare memoria), manipolare e, infine, distruggere (liberare la memoria). Queste operazioni rappresentano un tassello fondamentale per la gestione di *liste*, *alberi* e *heap* e verranno ampiamente introdotte e descritte nella parte introduttiva dei relativi capitoli. Le soluzioni ne faranno quindi uso senza prestare particolare attenzione alla loro implementazione, ma tenendo sempre bene in mente il loro funzionamento. Per aiutarci in questo possiamo fare uso della *documentazione*.[1] Se durante la scrittura di una soluzione, ad esempio, non si ricordano i parametri di una determinata primitiva o il suo funzionamento è bene affidarsi alla descrizione riportata all'inizio del capitolo o alla pagina web che raccoglie la documentazione di quella particolare struttura dati. I link sono riportati nei capitoli a seguire e sono anche raccolti nella pagina associata a questo libro: https://federicobolelli.it/pages/book-pubs.html.

Alcuni esercizi richiedono di implementare algoritmi che, pur non rappresentando funzioni primitive, sono utili (e vengono riusati) per la risoluzione di problemi più complessi. Questi esercizi sono marcati, sia nell'indice che nel titolo, con l'icona 🏢. Questo approccio ci consente di evitare ridondanza nel testo, senza perdere di generalità o di chiarezza.

Alcuni degli algoritmi che si chiede di implementare vengono spesso utilizzati per valutare le competenze tecnico-teoriche dei candidati durante i colloqui di lavoro per grandi colossi come Amazon, Facebook, Google, Microsoft e altre aziende più o meno importanti. Questi esercizi, tradotti e riadattati per rispondere alle esigenze del testo, sono identificati con l'icona 👤.

Altri esercizi ancora, identificati con l'icona 📘, contengono, nella descrizione della soluzione, una spiegazione di elementi teorici relativi all'argomento trattato. Il motivo per cui questi concetti vengono presentati nella soluzione di un esercizio piuttosto che nei capitoli introduttivi è semplice: la spiegazione su esempi è spesso più chiara ed efficace di quella a parole.

Quindi, come va usato questo libro? Le soluzioni sono volutamente separate dal testo del problema e riportate nel Capitolo 7. Una volta lette le specifiche di un esercizio e analizzato il problema dovresti aprire il tuo IDE, creare i file `.c` e/o `.h` richiesti dal testo e aggiungerne uno ulteriore per la funzione `main()` che, in questo libro, chiameremo sempre `main.c`. Scrivere la funzione `main()` è indispensabile per verificare il funzionamento della soluzione implementata.

Lo scopo delle soluzioni riportate in fondo al libro è quello di *sbloccare* lo studente in difficoltà, ma anche quello di mostrare approcci alternativi a quelli da lui seguiti. Tutto il

[1] La documentazione del software è un testo scritto che accompagna il codice descrivendolo a parole. Per una funzione, ad esempio, la documentazione riporterà il significato dei parametri di input e di eventuali parametri di output o dei valori di ritorno, nonché una descrizione dell'effetto che l'invocazione di questa funzione avrà sull'input e, soprattutto, sull'output.

codice riportato in questo libro è stato opportunamente testato e importato da progetti che compilano, questo significa che dovrebbe essere privo di errori, almeno nei casi considerati: ci scusiamo sin d'ora di eventuali *bug* sfuggiti ai nostri test, chiedendo aiuto a voi lettori nel segnalarceli.

1.2 Scelte di Stile

In questo paragrafo vengono illustrate, motivate e spiegate le scelte di stile adottate nel seguito della trattazione.

1.2.1 *Google C++ Style Guide*

Partiamo dalla forma. Tutto il codice riportato in questo libro è stato scritto seguendo le linee guida di stile di Google per il linguaggio C++. A cosa servono queste guide di stile? Perché utilizzare quelle del C++ se il codice è scritto in C?

Cerchiamo di chiarire questi dubbi rispondendo alle domande. Partiamo dalla seconda, la più semplice:

1. La *Google C Style Guide* non esiste;
2. Come dice lo stesso ideatore del linguaggio C++, Bjarne Stroustrup, "C++ is a direct descendant of C that retains almost all of C as a subset.".[2]

Con una semplice ricerca sul web ti accorgerai che in realtà esistono numerose linee guida di stile specifiche per il linguaggio C. A nostro avviso nessuna di queste è efficace e soprattutto diffusa come quella di Google.

Ma cosa sono queste guide di stile? Un insieme di regole che definiscono lo stile del nostro codice garantendo una miglior leggibilità e manutenibilità. Attenzione a non confondere le regole di stile con quelle di sintassi, le prime sono linee guida che il programmatore può deliberatamente scegliere di non seguire, mentre le altre devono essere sempre rispettate perché il codice compili. Definire una funzione o una variabile usando casualmente lettere maiuscole o minuscole non pregiudicherà la correttezza del programma, omettere il `;` dopo uno *statement*, invece, è proprio sbagliato!

Perché devo ricordarmi delle regole in più? Le motivazioni sono veramente tante, ma vediamo assieme le più importanti o comunque quelle più evidenti e significative dal punto di vista di questo testo:

1. Facilità di lettura e mantenimento del codice. Avere una convenzione per i nomi ci permette ad esempio di distinguere immediatamente e facilmente le funzioni (`ExampleFunction`) dalle variabili (`example_variabile`), dalle *macro* (`EXAMPLE_MACRO`) e dagli *enum* (`kExampleEnumField`);
2. Durante la scrittura del codice, avere uno stile predefinito riduce il numero di scelte che lo sviluppatore deve fare, permettendo quindi di concentrarsi sulla logica del programma invece che sulle decisioni di stile.

Quali sono queste regole? Le regole sono veramente tante,[3] noi ci soffermeremo sui principali costrutti relativi al C.

[2] https://www.stroustrup.com/bs_faq.html#difference.
[3] https://google.github.io/styleguide/cppguide.html#C++_Version.

Capitolo 2

Ricorsione

2.1 Un Breve Ripasso

Nella logica matematica e nell'informatica, le funzioni ricorsive sono una classe di funzioni dai numeri naturali ai numeri naturali che sono *calcolabili* attraverso funzioni di base e regole costruttive che usano la funzione stessa.

Dato un programma, una funzione si può definire ricorsiva se invoca sé stessa direttamente o indirettamente. Nel primo caso parleremo di *ricorsione diretta*, mentre nel secondo di *ricorsione indiretta* o *mutua ricorsione*. Un processo o algoritmo che è espresso in termini di sé stesso, ovvero che fa uso di chiamate a funzione ricorsive, si definisce ricorsivo. La ricorsione si definisce *lineare* quando vi è solo una chiamata ricorsiva all'interno della definizione della funzione stessa, viceversa si definisce ricorsione *non lineare* nel caso in cui le chiamate ricorsive siano più di una.

Gran parte delle funzioni ricorsive, ed in particolare le ricorsive lineari, possono essere realizzate facendo uso della sola iterazione. È bene però sottolineare che in alcuni casi la soluzione ricorsiva richiede l'utilizzo di strutture dati meno articolate e risulta più semplice di quella iterativa. Gli algoritmi di *backtracking* e le visite degli *alberi*, che vedremo nei prossimi capitoli, ne sono un esempio lampante. Per tale motivo è fondamentale padroneggiare il concetto di ricorsione ed essere in grado di risolvere problemi facendo uso di algoritmi ricorsivi. In questo capitolo vi proponiamo numerosi esercizi di difficoltà variabile, tutti risolti utilizzando la ricorsione.

Prima di procedere è bene però ripassare alcuni concetti fondamentali. Quando possibile, questi concetti verranno illustrati con riferimento al problema della somma dei numeri interi da `1` a `n`: `1 + 2 + 3 + 4 + 5 + ... + (n - 1) + n` la cui soluzione iterativa è data dal seguente ciclo[1]:

```
int sum = 0;
for (int i = 1; i <= n; ++i) {
```

[1] Si noti che il problema della somma dei primi `n` numeri naturali può essere risolto, senza l'uso dell'iterazione e della ricorsione, con la formula di Gauss: $\sum_{k=1}^{n} k = \frac{n \cdot (n+1)}{2}$.

```
    }

    // Aggiornamento dei parametri della chiamata ricorsiva e
    // sostituzione della chiamata ricorsiva con il goto.
    s = s + n;
    n = n - 1;
    goto begin;
}
```

Quando è in grado di farlo, il compilatore cerca sempre di convertire la ricorsione in iterazione, così da minimizzare i requisiti in termini di spazio e tempo. I programmi ricorsivi hanno requisiti di spazio maggiori rispetto a equivalenti programmi iterativi, in quanto tutte le funzioni rimangono nello *stack* fino a quando non viene raggiunto uno dei casi base. Lo stesso vale per i requisiti di tempo: la chiamata a funzione e la gestione dei valori di ritorno introduce un costo addizionale in termini di tempo di esecuzione. Ma quindi perché usare la ricorsione al posto dell'iterazione? Come già specificato all'inizio di questa introduzione, esistono numerosi problemi intrinsecamente ricorsivi la cui implementazione iterativa risulterebbe complessa ed articolata.[2] In questi casi è estremamente più semplice implementare soluzioni ricorsive e lasciare che sia il compilatore, se e quando se ne accorge, ad eseguire la conversione.

2.2 Esercizi

2.2.1 Esercizio – fattoriale

Il fattoriale di un numero n, indicato con $n!$, è definito, per $n \geq 0$, come

$$n! = \begin{cases} 1, & se\ n = 0 \\ \prod_{k=1}^{n} k, & se\ n > 0 \end{cases}$$

Nel file `fattoriale.c` implementare in linguaggio C la funzione corrispondente alla seguente dichiarazione:

```
extern double Fattoriale(int n);
```

La funzione restituisce il fattoriale di `n` calcolato ricorsivamente. Se `n` è negativo la funzione restituisce `-1`. Non è consentito l'uso di librerie esterne. Utilizzare internamente `double` per tutti i calcoli, in modo tale da avere una precisione sufficiente a contenere il risultato.

2.2.2 Esercizio – binomiale

Nel file `binomiale.c` implementare la definizione della funzione:

```
double Binomiale(unsigned int n, unsigned int k);
```

La funzione accetta come parametri due `unsigned int` e restituisce il coefficiente binomiale corrispondente, calcolato in maniera ricorsiva:

[2] Se l'obiettivo è quello di avere un'implementazione efficiente per la risoluzione di un determinato problema è bene, quando possibile, optare per soluzioni iterative, anche se più complesse e/o articolate.

Capitolo 7

Soluzioni

Soluzione dell'esercizio 2.2.1 – fattoriale

Per la soluzione di questo esercizio utilizzeremo una funzione ausiliaria `FattorialeRec()`.

La funzione `Fattoriale()` (non ricorsiva) controlla il valore di input `n`:

1. Se `n < 0` siamo di fronte ad un input non corretto e la funzione ritorna `-1` come specificato nel testo;
2. In tutti gli altri casi (`n >= 0`) la funzione ritorna il risultato della chiamata alla funzione ricorsiva `FattorialeRec()`.

La funzione `FattorialRec` (ricorsiva) è responsabile del calcolo del fattoriale di `n`. Sappiamo già che `n` è positivo o al più nullo, quindi non dobbiamo preoccuparci di controllare l'input. Per prima cosa, essendo la funzione ricorsiva, dobbiamo determinare il/i caso/i base. In questo ci viene in aiuto la definizione stessa del problema:

$$n! = \begin{cases} 1, & se\ n = 0 \\ \prod_{k=1}^{n} k, & se\ n > 0 \end{cases}$$

il caso base della formula si ha per `n = 0`: la ricorsione termina ritornando il valore `1` in quanto `0! = 1`. Nel caso in cui `n > 0`, invece, non possiamo stabilire a priori il valore di `n! = n*(n-1)!`, ma dobbiamo prima di tutto calcolare il valore di `(n-1)!`. Per tale motivo chiamiamo la funzione ricorsiva `FattorialeRec()` a cui passiamo appunto `n-1`. Una volta calcolato `(n-1)!` la funzione ritornerà `n*(n-1)!`. Lo stesso procedimento viene eseguito fino a quando non si incontra il caso base `n = 0`.

Quindi, seguendo il procedimento a ritroso per `n = 5` si ha che:

1. `FattorialeRec(0)` ritorna `1`
2. `FattorialeRec(1)` ritorna `1 * FattorialeRec(0)`, ovvero `1*1 = 1`
3. `FattorialeRec(2)` ritorna `2 * FattorialeRec(1)`, ovvero `2*1 = 2`
4. `FattorialeRec(3)` ritorna `3 * FattorialeRec(2)`, ovvero `3*2 = 6`
5. `FattorialeRec(4)` ritorna `4 * FattorialeRec(3)`, ovvero `4*6 = 24`

6. `FattorialeRec(5)` ritorna `5 * FattorialeRec(4)`, ovvero `5*24 = 120`

Ad un primo sguardo la funzione `FattorialeRec()` potrebbe sembrare una ricorsione di coda (*tail recursion*), ma in realtà non lo è in quanto l'ultima operazione che viene eseguita è il prodotto tra `n` e il risultato della chiamata ricorsiva.

Si noti che il controllo dell'input (`if (n < 0) { return -1 }`) poteva essere inserito direttamente nella funzione ricorsiva, evitando quindi la necessità di una funzione ausiliaria: così facendo il controllo verrebbe però inutilmente eseguito ad ogni passo della ricorsione.

```
/* fattoriale.c */
static double FattorialeRec(int n) {
    // Verifica del caso base
    if (n == 0) {
        return 1;
    }

    // Chiamata ricorsiva
    return n * FattorialeRec(n - 1);
}

double Fattoriale(int n) {
    // Verifica dei parametri di input
    if (n < 0) {
        return -1;
    }

    return FattorialeRec(n);
}
```

Un lettore curioso potrebbe a questo punto chiedersi come trasformare la funzione `FattorialeRec()` affinché diventi una ricorsione di coda. Per imporre che la chiamata ricorsiva sia l'ultima operazione eseguita dalla funzione stessa dobbiamo salvare il risultato parziale in un parametro aggiuntivo della chiamata, `r`. Come specificato anche nel testo dell'esercizio, è opportuno che i risultati intermedi siano memorizzati in un `double`.

In questo modo, raggiunto il caso base la funzione potrà ritornare il risultato del fattoriale, memorizzato in `r`. Ovviamente la funzione `FattorialeRec()` dovrà essere invocata la prima volta con `r = 1`.

L'implementazione *tail recursive* è riportata di seguito.

```
/* fattoriale.c */
static double FattorialeRec(int n, double r) {
    // Verifica del caso base
    if (n == 0) {
        return r;
    }

    // Chiamata ricorsiva
    return FattorialeRec(n - 1, n * r);
```

```
}

double Fattoriale(int n) {
    // Verifica dei parametri di input
    if (n < 0) {
        return -1;
    }

    return FattorialeRec(n, 1);
}
```

Main

Il `main()` è semplicemente una serie di chiamate alla funzione dichiarata all'inizio. Si noti che il fattoriale cresce molto rapidamente all'aumentare di `n`. Sapreste dire qual è il più grande numero `n` il cui fattoriale è rappresentabile in un `int` a `32 bit`? E in un `double`?

```
/* main.c */
extern double Fattoriale(int n);

int main(void) {
    double f;

    f = Fattoriale(-5); // -1.0
    f = Fattoriale( 0); // 1.0
    f = Fattoriale( 1); // 1.0
    f = Fattoriale( 3); // 6.0
    f = Fattoriale( 5); // 120.0
    f = Fattoriale(10); // 3628800.0
    f = Fattoriale(15); // 1307674368000.0

    return 0;
}
```

Soluzione dell'esercizio 2.2.2 – binomiale

La soluzione presentata utilizza una funzione ausiliaria `BinomialeRec()`, i cui parametri di input sono gli stessi della funzione `Binomiale()`. Come nell'esercizio sul fattoriale la funzione di "base" non è ricorsiva e si occupa semplicemente di verificare i casi particolari `n = 0` e `k > n`, ovvero i casi in cui il calcolo del binomiale è privo di significato. Come specificato dal testo, se una di queste condizioni si verifica la funzione termina restituendo `-1`, in caso contrario viene chiamata la funzione ausiliaria `BinomialeRec()` che calcola, ricorsivamente $\binom{n}{k}$ usando la formula. In questo esercizio abbiamo due casi base: `k = 0` e `k = n`.

```
/* binomiale.c */
static double BinomialeRec(int n, int k) {

    // Verifica dei casi base
```

Capitolo 8

Indice degli Esercizi per Tipologia

8.1 Ricorsione

Esercizio 2.2.1 – fattoriale 20
Esercizio 2.2.2 – binomiale 20
Esercizio 2.2.3 – fibonacci 21
Esercizio 2.2.4 – sin-x 21
Esercizio 2.2.5 – newton 21
Esercizio 2.2.6 – secanti 22
Esercizio 2.2.7 – trailing 23
Esercizio 2.2.8 – palindromo 23
Esercizio 2.2.9 – hailstone 24
Esercizio 2.2.10 – inverti-array 24
Esercizio 2.2.11 – atoi 24
Esercizio 2.2.12 – power-two 25
Esercizio 2.2.13 – quoziente 25
Esercizio 2.2.14 – inverti-cifre 26
Esercizio 2.2.15 – MCD 26
Esercizio 2.2.16 – recaman 26
Esercizio 2.2.17 – frazione-egizia 27
Esercizio 3.2.1 – stringhe 32
Esercizio 3.2.2 – stringhe-mod 33
Esercizio 3.2.3 – subset-equal 33
Esercizio 3.2.4 – regole 33
Esercizio 3.2.5 – gruppi 34
Esercizio 3.2.6 – gruppi-norep 35
Esercizio 3.2.7 – n-steps 36
Esercizio 3.2.8 – articoli 36
Esercizio 3.2.9 – colormap 37
Esercizio 3.2.10 – cacciatore-preda 38

Esercizio 3.2.11 – gola-cresta 39
Esercizio 3.2.12 – alpine-road 40
Esercizio 3.2.13 – robot-grid 41
Esercizio 3.2.14 – robot-grid-mod 43
Esercizio 3.2.15 – queens 43
Esercizio 3.2.16 – nonograms 45
Esercizio 4.3.1 – length 57
Esercizio 4.3.2 – occorrenze 58
Esercizio 4.3.9 – filter 60
Esercizio 4.3.11 – vette 61
Esercizio 4.3.12 – bucket-sort 61
Esercizio 4.3.15 – insert-ord 63
Esercizio 4.3.18 – ordered-merge 65
Esercizio 4.3.19 – remove-dup 65
Esercizio 5.2.1 – preorder-delete 75
Esercizio 5.2.2 – mirror 75
Esercizio 5.2.3 – compare 76
Esercizio 5.2.4 – height 76
Esercizio 5.2.5 – read 76
Esercizio 5.2.6 – write 78
Esercizio 5.2.8 – tree2bst 78
Esercizio 5.2.9 – avl 79
Esercizio 5.2.10 – vector2tree 80
Esercizio 5.2.11 – count 81
Esercizio 5.2.12 – prune 81
Esercizio 5.2.13 – mediano 82
Esercizio 5.2.15 – pred-succ 83
Esercizio 5.2.16 – level-order 84
Esercizio 5.2.17 – isomorfi 85
Esercizio 5.2.18 – equivalenti 85
Esercizio 5.2.19 – diameter 86

8.2 Backtracking

Esercizio 3.2.1 – stringhe 32
Esercizio 3.2.2 – stringhe-mod 33
Esercizio 3.2.3 – subset-equal 33
Esercizio 3.2.4 – regole 33
Esercizio 3.2.5 – gruppi 34
Esercizio 3.2.6 – gruppi-norep 35
Esercizio 3.2.7 – n-steps 36
Esercizio 3.2.8 – articoli 36
Esercizio 3.2.9 – colormap 37
Esercizio 3.2.10 – cacciatore-preda 38
Esercizio 3.2.11 – gola-cresta 39
Esercizio 3.2.12 – alpine-road 40
Esercizio 3.2.13 – robot-grid 41

Esercizio 3.2.14 – robot-grid-mod 43
Esercizio 3.2.15 – queens 43
Esercizio 3.2.16 – nonograms 45

8.3 Liste

Esercizio 4.3.1 – length 57
Esercizio 4.3.2 – occorrenze 58
Esercizio 4.3.3 – coords-load 58
Esercizio 4.3.4 – vector-list 58
Esercizio 4.3.5 – taglia 59
Esercizio 4.3.6 – split 59
Esercizio 4.3.7 – concatena 60
Esercizio 4.3.8 – concatena-n 60
Esercizio 4.3.9 – filter 60
Esercizio 4.3.10 – any-loop 61
Esercizio 4.3.11 – vette 61
Esercizio 4.3.12 – bucket-sort 61
Esercizio 4.3.13 – common-tail 63
Esercizio 4.3.14 – back2front 63
Esercizio 4.3.15 – insert-ord 63
Esercizio 4.3.16 – sum 64
Esercizio 4.3.17 – insertion-sort 64
Esercizio 4.3.18 – ordered-merge 65
Esercizio 4.3.19 – remove-dup 65

8.4 Alberi

Esercizio 5.2.1 – preorder-delete 75
Esercizio 5.2.2 – mirror 75
Esercizio 5.2.3 – compare 76
Esercizio 5.2.4 – height 76
Esercizio 5.2.5 – read 76
Esercizio 5.2.6 – write 78
Esercizio 5.2.7 – vector2bst 78
Esercizio 5.2.8 – tree2bst 78
Esercizio 5.2.9 – avl 79
Esercizio 5.2.10 – vector2tree 80
Esercizio 5.2.11 – count 81
Esercizio 5.2.12 – prune 81
Esercizio 5.2.13 – mediano 82
Esercizio 5.2.14 – morris 82
Esercizio 5.2.15 – pred-succ 83
Esercizio 5.2.16 – level-order 84
Esercizio 5.2.17 – isomorfi 85

Esercizio 5.2.18 – equivalenti ... 85
Esercizio 5.2.19 – diameter ... 86

8.5 BST

Esercizio 5.2.7 – vector2bst ... 78
Esercizio 5.2.8 – tree2bst ... 78
Esercizio 5.2.9 – avl ... 79
Esercizio 5.2.15 – pred-succ ... 83

8.6 Heap

Esercizio 6.2.1 – heapify ... 95
Esercizio 6.2.2 – remove ... 95
Esercizio 6.2.3 – copy ... 95
Esercizio 6.2.4 – is-heap ... 95
Esercizio 6.2.5 – get-pop ... 96
Esercizio 6.2.6 – heapsort ... 96
Esercizio 6.2.7 – kth-least ... 96
Esercizio 6.2.8 – merge ... 96
Esercizio 6.2.9 – min2max ... 97

8.7 Input/Output con File

Esercizio 3.2.5 – gruppi ... 34
Esercizio 3.2.6 – gruppi-norep ... 35
Esercizio 4.3.3 – coords-load ... 58
Esercizio 5.2.5 – read ... 76
Esercizio 5.2.6 – write ... 78

8.8 Algoritmi di Ordinamento

Esercizio 4.3.12 – bucket-sort ... 61
Esercizio 4.3.15 – insert-ord ... 63
Esercizio 4.3.17 – insertion-sort ... 64
Esercizio 4.3.18 – ordered-merge ... 65
Esercizio 4.3.19 – remove-dup ... 65
Esercizio 6.2.6 – heapsort ... 96

8.9 Colloquio di Lavoro - Interview

Esercizio 2.2.1 – fattoriale 20
Esercizio 3.2.7 – n-steps 36
Esercizio 3.2.9 – colormap 37
Esercizio 3.2.13 – robot-grid 41
Esercizio 3.2.15 – queens 43
Esercizio 4.3.10 – any-loop 61
Esercizio 4.3.16 – sum 64
Esercizio 4.3.19 – remove-dup 65
Esercizio 5.2.2 – mirror 75
Esercizio 5.2.9 – avl 79
Esercizio 5.2.15 – pred-succ 83
Esercizio 5.2.17 – isomorfi 85

8.10 Elementi di Teoria

Esercizio 2.2.3 – fibonacci 21
Esercizio 5.2.9 – avl 79
Esercizio 5.2.14 – morris 82
Esercizio 5.2.16 – level-order 84
Esercizio 6.2.1 – heapify 95