Introduzione alla programmazione in C

Appunti delle lezioni di Tecniche di programmazione

Giorgio Grisetti
Luca Iocchi
Daniele Nardi
Fabio Patrizi
Alberto Pretto

Dipartimento di Ingegneria Informatica, Automatica e Gestionale
Facoltà di Ingegneria dell'Informazione, Informatica, Statistica
Università di Roma "La Sapienza"

Edizione 2020/2021

2021

Indice

8. Ricorsione

8.1. Funzioni ricorsive

Una funzione si dice *ricorsiva* se al suo interno contiene un'attivazione di sè stessa (eventualmente indirettamente, attraverso l'attivazione di altre funzioni). Vediamo alcuni esempi di funzioni matematiche sui naturali definite in modo ricorsivo.

Esempio: fattoriale

$$fatt(n) = \begin{cases} 1, & se\ n = 0 \quad (\texttt{caso base}) \\ n \cdot fatt(n-1), & se\ n > 0 \quad (\texttt{caso ricorsivo}) \end{cases}$$

La definizione ricorsiva di una funzione ha le seguenti caratteristiche:

- uno (o più) *casi base*, per i quali il risultato può essere determinato direttamente;
- uno (o più) *casi ricorsivi*, per i quali si riconduce il calcolo del risultato al calcolo della stessa funzione su un valore più piccolo o più semplice.

La definizione ricorsiva del fattoriale può essere implementata direttamente attraverso una funzione ricorsiva.

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

8.1.1. Esempio: implementazione ricorsiva della somma di due interi

Sfruttiamo la seguente definizione ricorsiva di somma tra due interi non negativi:

$$somma(x,y) = \begin{cases} x, & se\ y = 0 \\ 1 + somma(x, y-1), & se\ y > 0 \end{cases}$$

```
int somma(int x, int y) {
  if (y == 0)
    return x;
  else
    return 1 + somma(x, y-1);
}
```

8.1.2. Esempio: implementazione ricorsiva del prodotto tra due interi

Sfruttiamo la seguente definizione ricorsiva del prodotto tra due interi non negativi:

$$prodotto(x,y) = \begin{cases} 0, & se\ y = 0 \\ somma(x, prodotto(x, y-1)), & se\ y > 0 \end{cases}$$

Implementazione:

```
int prodotto(int x, int y) {
  if (y == 0)
    return 0;
  else
    return somma(x, prodotto(x, y-1));
}
```

8.1.3. Esempio: implementazione ricorsiva dell'elevamento a potenza

Sfruttiamo la seguente definizione ricorsiva dell'elevamento a potenza tra due interi non negativi:

$$potenza(b,e) = \begin{cases} 1, & se\ e = 0 \\ prodotto(b, potenza(b, e-1)), & se\ e > 0 \end{cases}$$

Implementazione:

```
int potenza(int b, int e) {
  if (e == 0)
    return 1;
  else
    return (prodotto(b, potenza(b, e-1)));
}
```

8.2. Confronto tra ricorsione e iterazione

Le funzioni implementate in modo ricorsivo ammettono anche un'implementazione iterativa, come già visto per somma, prodotto e potenza.

Esempio: implementazione iterativa del fattoriale, sfruttando la seguente definizione iterativa:

$$fatt(n) = n \cdot (n-1) \cdot (n-2) \cdot \cdots \cdot 2 \cdot 1$$

```
int fattorialeIterativa(int n) {
  int ris = 1;
  while (n > 0) {
    ris = ris * n;
    n--;
  }
  return ris;
}
```

Caratteristiche dell'implementazione iterativa:

- inizializzazione:
  Es. ris = 1;
- operazione del ciclo, eseguita un numero di volte pari al numero di ripetizioni del ciclo:
  Es. ris = ris * n;

- terminazione:
  Es. n- -; consente di rendere la condizione (n > 0) del ciclo while falsa

Implementazione ricorsiva, sfruttando la definizione ricorsiva data precedentemente:

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

Caratteristiche dell'implementazione ricorsiva:

- passo base:
  Es. return 1;
- passo ricorsivo:
  Es. return n * fattoriale(n - 1) ;
- terminazione è garantita dal fatto che la chiamata ricorsiva fattoriale(n-1) diminuisce di uno il valore passato come parametro, per cui, se inizialmente n>0, prima o poi si arriverà ad un'attivazione in cui la condizione n==0 sarà vera e quindi viene eseguito solo il passo base.

8.2.1. Confronto tra ciclo di lettura e lettura ricorsiva

Struttura del ciclo di lettura per i file:

```
leggi primo elemento
while (elemento valido) {
  elabora elemento letto;
  leggi elemento successivo;
}
```

Struttura della lettura ricorsiva per i file:

```
leggi un elemento
if (elemento valido) {
  elabora elemento letto;
```

```
  chiama ricorsivamente lettura;
}
```

Struttura della lettura ricorsiva per tipi indicizzati generici:

```
if (dato vuoto) {
  passo base;
}
else {
  leggi un elemento
  elabora elemento letto;
  chiama ricorsivamente lettura sui dati rimanenti;
}
```

Esempio: copia di un file di input in un file di output.
Implementazione iterativa:

```
void copiaIterativa(FILE *i, FILE *o) {
    char x;
    x = fgetc(i);
    while (x != EOF) {
        fputc(x,o);
        x = fgetc(i);
    }
}
```

Implementazione ricorsiva:

```
void copiaRicorsiva(FILE * i, FILE * o) {
    char x;
    x = fgetc(i);
    if (x != EOF) {
        fputc(x,o);
        copiaRicorsiva(i,o);
    }
}
```

8.2.2. Esempio: gli ultimi saranno i primi

Vogliamo leggere i caratteri di un file di input e copiarli su un file di output, invertendone l'ordine. Facendo uso della ricorsione, questa operazione risulta particolarmente semplice.

```
void copiaInversa(FILE *i, FILE *o) {
    char x;
    x = fgetc(i);
    if (x != EOF) {
        copiaInversa(i,o);
        fputc(x,o);
    }
}
```

La funzione copiaInversa non si può facilmente formulare in modo iterativo, in quanto la scrittura su file in ordine inverso richiederebbe la memorizzazione dei caratteri letti in una struttura dati apposita. Analizzeremo questo esempio più avanti, mostrando in che modo le zone di memoria associate alle variabili locali delle attivazioni ricorsive fungono da memoria temporanea per i caratteri letti in input.

Facciamo notare la differenza tra questo tipo di ricorsione ed i casi più semplici visti in precedenza, in cui passare ad un'implementazione iterativa risulta immediato, come nel caso della funzione copiaRicorsiva. Questi casi sono quelli in cui l'ultima istruzione effettuata prima che la funzione termini è la chiamata ricorsiva (*tail recursion*). Alcuni compilatori sono in grado di riconoscere i casi di tail recursion, e di effetuare delle ottimizzazioni per generare un codice macchina più efficiente.

Facciamo invece notare che, in generale, un'implementazione ricorsiva potrebbe risultare più inefficiente di una corrispondente implementazione iterativa, a causa della necessità di gestire le chiamate ricorsive, come mostrato più avanti.

8.3. Schemi di ricorsione

Esistono diversi schemi ricorsivi degli algoritmi di base. Alcuni di essi sono illustrati di seguito.

8.3.1. Conteggio di elementi usando la ricorsione

Struttura della funzione ricorsiva per i file:

```
leggi un elemento;
if (!elemento valido)
   return 0;
else
```

```
   return 1 + risultato ricorsione;
```

Struttura della funzione ricorsiva per tipi indicizzati generici:

```
if (dato vuoto)
   return 0;
else
   return 1 + risultato ricorsione sui dati rimanenti;
```

8.3.1.1. Esempio: lunghezza di una sequenza di caratteri

Caratterizzazione ricorsiva dell'operazione di contare i caratteri in un file di input *i*:

- se *i* è vuoto, restituisci 0;
- altrimenti, leggi il primo carattere in *i* e restituisci 1 più il numero di caratteri presenti nel resto del file *i*.

```
int contaCaratteri(FILE *i) {
    char c;
    c = fgetc(i);
    if (c == EOF)
        return 0;
    else
        return 1 + contaCaratteri(i);
}
```

8.3.2. Conteggio condizionato di elementi usando la ricorsione

Struttura della funzione ricorsiva per i file:

```
leggi un elemento;
if (!elemento valido)
   return 0;
else if (condizione)
   return 1 + risultato-della-chiamata-ricorsiva;
else
   return risultato-della-chiamata-ricorsiva;
```

Struttura della funzione ricorsiva per tipi indicizzati generici:

```
if (dato vuoto)
   return 0;
else if (condizione)
   return 1 + risultato ricorsione sui dati rimanenti;
else
   return risultato ricorsione sui dati rimanenti;
```

8.3.2.1. Esempio: numero di occorrenze di un carattere in un file

Caratterizzazione ricorsiva dell'operazione di contare le occorrenze di un carattere *c* nel file di input *i*:

- se *i* è vuoto, restituisci 0;
- altrimenti, se il primo carattere di *i* è uguale a *c*, restituisci 1 più il numero di occorrenze di *c* nel resto del file *i*;
- altrimenti (ovvero se il primo carattere di *i* è diverso da *c*), allora restituisci il numero di occorrenze di *c* nel resto del file *i*.

```
int contaOccorrenzeCarattere(FILE *i, char x) {
    char c;
    c = fgetc(i);
    if (c == EOF)
        return 0;
    else if (c==x)
        return 1 + contaOccorrenzeCarattere(i,x);
    else
        return contaOccorrenzeCarattere(i,x);
}
```

8.3.3. Accumulazione usando la ricorsione

Supponiamo di voler effettuare un'operazione (ad esempio, la somma) tra tutti gli elementi di una collezione.

Struttura della funzione ricorsiva per i file:

```
leggi un elemento;
if (!elemento valido)
   return elemento-neutro-di-op;
```

```
else
   return valore-elemento op risultato-della-chiamata-ricorsiva;
```

dove elemento-neutro-di-op è l'elemento neutro rispetto all'operazione da effettuare (ad esempio, 0 per la somma, 1 per il prodotto, ecc.).

Struttura della funzione ricorsiva per tipi indicizzati generici:

```
if (dato vuoto)
   return elemento-neutro-di-op;
else
   return valore primo elemento <op>
          risultato ricorsione sui dati rimanenti;
```

8.3.3.1. Esempio: somma di interi in un file di input

Caratterizzazione ricorsiva dell'operazione di sommare i valori letti da testiera:

- leggi un elemento *i*;
- se il file è terminato, restituisci 0;
- altrimenti, restituisci la somma tra il valore letto e la somma dei valori del resto del file.

```
int sommaValori(FILE *i) {
    int finefile;
    int v;
    finefile = fscanf(i,"%d", &v);
    if (finefile == EOF)
        return 0;
    else
        return v + sommaValori(i);
}
```

8.3.3.2. Esempio: presenza di un valore in un insieme

Caratterizzazione ricorsiva dell'operazione di trovare un valore in un insieme di valori letti da tastiera:

- leggi un elemento i;
- se il file è terminato, restituisci false;
- altrimenti, se i é il valore cercato, restituisci true;
- altrimenti procedi la ricerca nel resto del file.

```
int trovaValore(FILE *i, int x) {
    int v;
    int finefile;
    finefile = fscanf(i,"%d", &v);
    if (finefile == EOF)
        return 0;   // false
    else if (v==x)
        return 1;   // true
    else
        return trovaValore(i,x);
}
```

oppure

```
int trovaValore2(FILE *i, int x) {
    int v;
    int finefile;
    finefile = fscanf(i,"%d", &v);
    if (finefile == EOF)
        return 0;
    else
        return (v==x) || trovaValore2(i,x);
}
```

8.3.3.3. Esempio: massimo di interi positivi letti da file

Caratterizzazione ricorsiva dell'operazione di trovare il massimo tra i valori di un insieme di interi positivi:

- leggi un elemento i;
- se il file è terminato, restituisci 0;
- altrimenti,
    1. trova il massimo m tra i valori rimanenti nel file;
    2. restituisci il maggiore tra i ed m.

```
int massimo(FILE *i) {
    int v;
    int finefile;
    finefile = fscanf(i,"%d", &v);
    if (finefile == EOF)
        return 0;
    else {
        int m = massimo(i);
        if (m>v) return m;
        else return v;
        // oppure
        // return (m>v)?m:v;
    }
}
```

8.4. Evoluzione della pila dei RDA nel caso di funzioni ricorsive

Nel caso di funzioni ricorsive, i meccanismi con cui evolvono la pila dei RDA ed il program counter sono identici al caso di funzioni non ricorsive. Tuttavia, è importante sottolineare che un RDA è associato ad *un'attivazione di una funzione* e non ad una funzione.

Esempio: consideriamo la seguente funzione ricorsiva e la sua attivazione dalla funzione main:

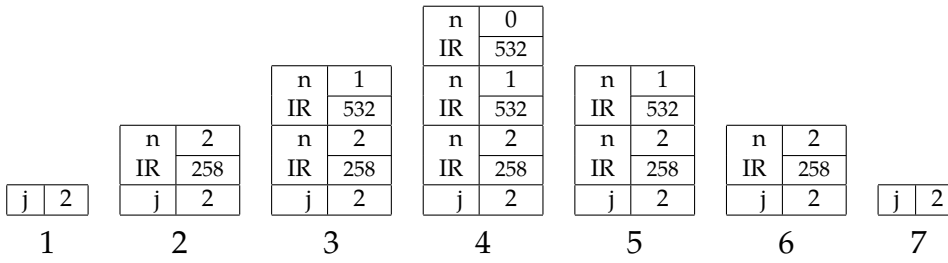
```
// funzione ausiliaria di stampa formattata
void stampa(int n, const char* str) {
    for (int i=0; i<10-n*2; i++) {
        printf(" ");
    }
    printf("ricorsivo(%d) - %s\n",n,str);
}

// funzione ricorsiva
void ricorsivo(int n) {
  if (n == 0) {
    stampa(n, "finito");
  }
  else {
    stampa(n,"attiva ricorsivo(n-1)");
    ricorsivo(n-1);
    stampa(n,"finito");
  }
}
```

```
int main() {
  int j;
  printf("Inserisci livello ricorsione\n");
  scanf("%d",&j);
  printf("Main - attiva ricorsivo(%d)\n",j);
  ricorsivo(j);
  printf("Main - Finito\n");
}
```

```
Inserisci livello ricorsione
3
Main - attiva ricorsivo(3)
    ricorsivo(3) - attiva ricorsivo(n-1)
      ricorsivo(2) - attiva ricorsivo(n-1)
        ricorsivo(1) - attiva ricorsivo(n-1)
          ricorsivo(0) - finito
        ricorsivo(1) - finito
      ricorsivo(2) - finito
    ricorsivo(3) - finito
Main - Finito
```

L'evoluzione della pila dei RDA per un livelo di ricersione 2 è mostrata qui di seguito. Abbiamo assunto che 258 sia l'indirizzo dell'istruzione che segue l'attivazione di ricorsivo(j) in main, e che 532 sia l'indirizzo dell'istruzione che segue l'attivazione di ricorsivo(n-1) in ricorsivo. Dal momento che le funzioni invocate non prevedono la restituzione di un valore di ritorno (il tipo di ritorno è void), i RDA non contengono una locazione di memoria per tale valore. Inoltre, non abbiamo indicato la funzione alla quale si riferisce ciascun RDA, in quanto il RDA in fondo alla pila è relativo a main, e tutti gli altri sono relativi ad attivazioni successive di ricorsivo.



Facciamo notare che per le diverse attivazioni ricorsive vengono creati diversi RDA sulla pila, con valori via via decrescenti del parametro

i, fino all'ultima attivazione ricorsiva, per la quale il parametro i assume valore 0. A questo punto non avviene più un'attivazione ricorsiva, viene stampato "finito", e l'attivazione termina. In cascata, avviene l'uscita dalle attivazioni precedenti, ogni volta preceduta dalla stampa di "ricorsivo(*i*) - finito".

Facciamo anche notare che codice associato alle diverse attivazioni ricorsive è sempre lo stesso, ovvero quello della funzione ricorsiva. Di conseguenza, l'indirizzo di ritorno memorizzato nei RDA per le diverse attivazioni ricorsive è sempre lo stesso (ovvero 532), tranne che per la prima attivazione, per la quale l'indirizzo di ritorno è quello di un'istruzione nella funzione main (ovvero 258).

8.5. Ricorsione multipla

Si ha **ricorsione multipla** quando un'attivazione di una funzione può causare *più di una attivazione ricorsiva* della stessa funzione.

Esempio: funzione ricorsiva per il calcolo dell'n-esimo numero di Fibonacci.

Fibonacci era un matematico pisano del 1200, interessato alla crescita di popolazioni. Ideò un modello matematico per stimare il numero di individui ad ogni generazione:

$F(n)$ ... numero di individui alla generazione n-esima

$$F(n) = \begin{cases} 0, & se\ n = 0 \\ 1, & se\ n = 1 \\ F(n-2) + F(n-1), & se\ n > 1 \end{cases}$$

$F(0)$, $F(1)$, $F(2)$, ... è detta sequenza dei numeri di Fibonacci, ed inizia con:
0, 1, 1, 2, 3, 5, 8, 13, 21, ...

Vediamo una funzione ricorsiva che, preso un intero positivo n, restituisce l'n-esimo numero di Fibonacci.

```
int fibonacci(int n) {
  if (n < 0) return -1;  // F(n) non e' definito per n
     negativo!
  if (n == 0)
    return 0;
  else if (n == 1)
    return 1;
  else
    return fibonacci(n-1) + fibonacci(n-2);
}
```

La chiamata ricorsiva attiverà due attivazioni della funzione ricorsiva.

8.5.1. Esempio: Torri di Hanoi

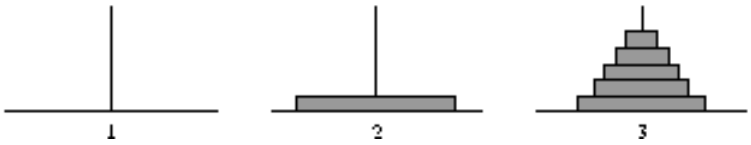
Il problema delle Torri di Hanoi ha origine da un'antica leggenda Vietnamita, secondo la quale un gruppo di monaci sta spostando una torre di 64 dischi (secondo la leggenda, quando i monaci avranno finito, verrà la fine del mondo). Lo spostamento della torre di dischi avviene secondo le seguenti regole:

- inizialmente, la torre di dischi di dimensione decrescente è posizionata su un perno 1;
- l'obiettivo è quello di spostarla su un perno 2, usando un perno 3 di appoggio;
- le condizioni per effettuare gli spostamenti sono:
  - tutti i dischi, tranne quello spostato, devono stare su una delle torri
  - è possibile spostare un solo disco alla volta, dalla cima di una torre alla cima di un'altra torre;
  - un disco non può mai stare su un disco più piccolo.

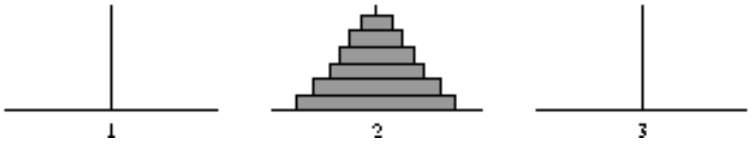
Lo stato iniziale (a), uno stato intermedio (b), e lo stato finale (c) per un insieme di 6 dischi sono mostrati nelle seguenti figure:

(a)

(b)



(c)



Vogliamo realizzare un programma che stampa la sequenza di spostamenti da fare. Per ogni spostamento vogliamo stampare un testo del tipo:

muovi un disco dal perno x al perno y

Idea: per spostare $n > 1$ dischi da 1 a 2, usando 3 come appoggio:

1. sposta $n - 1$ dischi da 1 a 3
2. sposta l'n-esimo disco da 1 a 2
3. sposta $n - 1$ dischi da 3 a 2

```
void muoviUnDisco(int sorg, int dest) {
    printf("muovi un disco da %d a %d\n", sorg, dest);
}

void muovi(int n, int sorg, int dest, int aux) {
    if (n == 1)
      muoviUnDisco(sorg, dest);
    else {
      muovi(n-1, sorg, aux, dest);
      muoviUnDisco(sorg, dest);
      muovi(n-1, aux, dest, sorg);
    }
  }
```

```
int main () {
    printf("Quanti dischi vuoi muovere?\n");
    int n;
    scanf("%d",&n);
    printf("Per muovere %d dischi da 1 a 2 con 3 come
    appoggio:\n",n);
    muovi(n, 1, 2, 3);
}
```

8.5.1.1. Numero di attivazioni nel caso di ricorsione multipla

Quando si usa la ricorsione multipla, bisogna tenere presente che il numero di attivazioni ricorsive potrebbe essere *esponenziale* nella profondità delle chiamate ricorsive (cioè nell'altezza massima della pila dei RDA).

Esempio: Torri di Hanoi

$$\begin{aligned} att(n) &= \text{numero di attivazioni di muoviUnDisco per } n \text{ dischi} \\ &= \text{numero di spostamenti di un disco} \end{aligned}$$

$$att(n) = \begin{cases} 1, & se\ n = 1 \\ 1 + 2 \cdot att(n-1), & se\ n > 1 \end{cases}$$

Senza "$1+$" nel caso di $n > 1$, avremmo $att(n) = 2^{n-1}$. Ne segue che $att(n) > 2^{n-1}$.

Si noti che nel caso del problema delle Torri di Hanoi il numero esponenziale di attivazioni è una caratteristica intrinseca del problema, nel senso che non esiste una soluzione migliore.

8.5.2. Esercizio: attraversamento di una palude

Si consideri un'area paludosa costituita da $R \times C$ zone quadrate, con R ed C noti, ognuna delle quali può essere una zona di terraferma (transitabile) o una zona di sabbia mobile (non transitabile). Ogni zona della palude è identificata da una coppia di coordinate $\langle r, c \rangle$, con $0 \leq r < R$ e $0 \leq c < C$. Diremo che r rappresenta la *riga* e c la *colonna* della zona $\langle r, c \rangle$. Per *passaggio* si intende una sequenza di zone di terraferma adiacenti che attraversano la palude da sinistra (colonna pari a 0) a destra (colonna pari ad $C - 1$). Siamo interessati ai passaggi in cui ad ogni passo ci si muove verso destra, per cui da una zona in colonna c si va ad una zona in colonna $c + 1$. In altre parole, la zona in posizione $\langle r, c \rangle$ si considera adiacente alle zone in posizione $\langle r - 1, c + 1 \rangle$, $\langle r, c + 1 \rangle$ e $\langle r + 1, c + 1 \rangle$, come mostrato nella seguente figura.

Nella figura che segue, il carattere '*' rappresenta una zona di terraferma mentre il carattere 'o' rappresenta una zona di sabbia mobile. La palude 1 è senza passaggi, mentre la palude 2 ha un passaggio (evidenziato).

	0	1	2	3	4	5
0	*	o	o	*	o	o
1	o	*	o	o	o	o
2	o	o	*	*	o	o
3	*	*	o	o	o	o
4	*	*	*	o	*	*

Palude 1

	0	1	2	3	4	5
0	*	o	o	*	o	o
1	*	o	o	o	o	o
2	o	*	o	o	o	*
3	o	o	*	*	*	o
4	o	*	o	o	o	o

Palude 2

Si richiede di verificare l'esistenza di almeno un passaggio e stamparlo se esiste (se ne esiste più di uno è sufficiente stampare il primo trovato).

8.5.2.1. Palude: rappresentazione di una palude

Per rappresentare una palude, definiamo un array di interi, e realizziamo un insieme di operazioni che consentono di utilizzarla:

- inizializzazione di una palude casuale, dati il numero di righe e di colonne, ed un valore reale compreso tra 0 e 1 che rappresenta la probabilità che una generica zona sia di terraferma;
- verifica se la zona di coordinate $\langle r, c \rangle$ è di terra;
- stampa la palude utilizzando i caratteri * e o per rappresentare le zone di terraferma e di sabbie mobili, rispettivamente.

```
int const righe = 10;
int const colonne = 10;
double probTerra = 0.5;

// dichiara la palude
int palude[righe][colonne];

// Operazioni sulla palude
int terra(int r, int c) {
    return (r >= 0) && (r < righe) &&
      (c >= 0) && (c < colonne) &&
      palude[r][c];
}
```

```
  void initPalude(double probTerra) {
     srand( time(NULL) );
     for (int r = 0; r < righe; r++)
        for (int c = 0; c < colonne; c++)
           palude[r][c] = rand()/(double)RAND_MAX <
    probTerra;
  }

void stampaPalude () {
    for (int r = 0; r < righe; r++) {
      for (int c = 0; c < colonne; c++)
         printf(palude[r][c]? "*" : "o");
      printf("\n");
    }
}
```

8.5.2.2. Palude: soluzione dell'attraversamento

La soluzione richiede di trovare una sequenza di zone della palude, in cui la prima posizione sia in colonna 1, mentre l'ultima sia in colonna C. Ogni posizione della sequenza deve essere adiacente alla successiva. Per esempio, se la prima posizione è $\langle 3, 0\rangle$, la seconda può essere $\langle 4, 1\rangle$, ma non $\langle 3, 2\rangle$. Dal momento che ad ogni passo dobbiamo muoverci verso destra, il percorso sarà lungo esattamente C passi.

Per esplorare la palude scegliamo di utilizzare un metodo ricorsivo. Questa scelta è la più intuitiva, dal momento che il processo di ricerca è inerentemente ricorsivo. L'algoritmo si può riassumere in questo modo: al primo passo si cerca una zona di terraferma nella prima colonna. Se c'è, si parte da quel punto. Al passo generico, ci si trova in una posizione $\langle r, c\rangle$. Se la posizione è di terraferma si può proseguire e si invoca ricorsivamente la ricerca sulle posizioni adiacenti, ovvero $\langle r-1, c+1\rangle$, $\langle r, c+1\rangle$ ed $\langle r+1, c+1\rangle$. Se invece la zona è di sabbia mobile non si può proseguire e la ricerca da quella zona termina. La ricerca termina quando si arriva ad una zona sull'ultima colonna, ovvero c coincide con $colonne - 1$ e questa zona è una zona di terraferma.

Il generico passo di ricerca può essere implementato attraverso il seguente metodo ricorsivo cercaCammino, che riceve come parametri le coordinate $\langle r, c\rangle$ della zona dalla quale cercare il cammino.

```
int cercaCammino(int r, int c) {
  if (coordinate <r,c> della zona di palude non valide
      || <r,c> è una zona di sabbie mobili)
    return false;
  else if (<r,c> è sul bordo destro della palude)
    return true;
  else
    return cercaCammino(r-1, c+1) ||
           cercaCammino(r  , c+1) ||
           cercaCammino(r+1, c+1);
}
```

La funzione cercaCammino verifica solo se esiste un cammino da una posizione generica $\langle r, c\rangle$ fino all'ultima colonna. Dal momento che sono validi i cammini da una qualsiasi posizione della prima colonna, è necessario richiamare questo metodo in successione sulle posizioni $\langle r, 0\rangle$ della prima colonna, fino a quando non si è trovato un cammi-

no oppure si è arrivati all'ultima riga. Questo viene fatto dal metodo attraversaPalude.

8.5.2.3. Palude: costruzione del cammino

Per rappresentare un cammino, facciamo notare che esso deve avere lunghezza pari al numero di colonne della palude, e che le zone attraversate dal cammino stanno su righe successive a partire da 0 fino ad arrivare alla colonna più a destra. Quindi, possiamo usare un array di *colonne* elementi interi, nel quale il valore del generico elemento di indice c è pari all'indice r di riga della posizione $\langle r, c \rangle$ attraversata dal cammino. Ad esempio il cammino evidenziato nella palude 2 di sopra è rappresentato dall'array {1,2,3,3,3,2}.

Nella seguente implementazione abbiamo scelto di aggiungere ai metodi cercaCammino e attraversaPalude un ulteriore parametro di tipo array di interi che rappresenta un cammino, e fare in modo che tali metodi facciano side-effect aggiornando in modo opportuno il cammino.

```
int cercaCammino(int r, int c, int camm[]) {
    if (!terra(r,c))
      return 0;
    else {
      camm[c] = r;
      if (c == colonne-1)
        return 1;
      else
        return cercaCammino(r-1, c+1, camm) ||
               cercaCammino(r,   c+1, camm) ||
               cercaCammino(r+1, c+1, camm);
    }
}
```

```
int attraversaPalude(int camm[]) {
   for (int r = 0; r < righe; r++)
      if (cercaCammino(r, 0, camm)) return 1;
   return 0;
}

int main() {
    initPalude();
    stampaPalude();
    int cammino [colonne];
    for (int c=0; c < colonne; c++) cammino[c] = 0;
    if (attraversaPalude(cammino))
        for (int c=0;  c < colonne; c++) printf("%d",
    cammino[c]);
    else
        printf("Cammino: cammino inesistente");
    printf("\n");
}
```