Funzioni Ricorsive

Algoritmi 1 – Lezione 3

A. Monti
Sapienza Università di Roma

Corso: Algoritmi 1

Indice

1. Introduzione e concetti di base
2. Complessità delle funzioni ricorsive
   - Complessità temporale
   - Complessità spaziale
3. Esempi di funzioni ricorsive
   - Somma di una lista
   - Potenza x^n
   - Palindromi
   - Permutazioni
   - Numeri di Fibonacci

Definizione di Funzioni Ricorsive

Una **funzione ricorsiva**, in informatica, è una funzione che si richiama direttamente (o indirettamente) al proprio interno per risolvere un problema.

Questa tecnica si basa su due elementi fondamentali:

- un **caso base**, che definisce una condizione di arresto e fornisce una soluzione immediata;
- un **passo ricorsivo**, in cui la funzione risolve il problema scomponendolo in **sottoproblemi di dimensione minore**.

Esempio Python: calcolo ricorsivo del fattoriale

```
def fattorialeR(n):
    if n == 0:
        return 1                          # caso base
    return n * fattorialeR(n - 1)         # passo ricorsivo
```

Equazioni di Ricorrenza

Quando un algoritmo è definito ricorsivamente, anche il suo tempo di esecuzione può essere espresso in forma ricorsiva.

```
def fattorialeR(n):                    # T(n) =
    if n == 0:                         #      O(1)
        return 1                       #      O(1)
    return n * fattorialeR(n-1)        #    O(1) + T(n-1)
```

Indicando con $T(n)$ il tempo di esecuzione sull'input n, otteniamo la seguente **equazione di ricorrenza**:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 0 \quad \text{(caso base)} \\ T(n-1) + \Theta(1) & \text{se } n > 0 \quad \text{(passo ricorsivo)} \end{cases}$$

Le **equazioni di ricorrenza** sono strumenti matematici utilizzati per descrivere il comportamento temporale (e talvolta anche spaziale) di algoritmi ricorsivi. Queste equazioni esprimono il tempo di esecuzione di un algoritmo in funzione del numero di sottoproblemi risolti ricorsivamente e dei costi associati a ciascuna chiamata.

Nel seguito, vedremo come risolvere tali equazioni per determinare esplicitamente la complessità dell'algoritmo.

Ricorsione vs Iterazione I: Aspetti tecnici

Ricorsione e **iterazione** sono due tecniche per risolvere problemi ripetitivi.

- **Ricorsione**: una funzione chiama sé stessa su versioni più semplici del problema.
- **Iterazione**: si basa su strutture di controllo come *for* e *while*.

Attenzione al caso base!

- Ogni funzione ricorsiva deve avere almeno un **caso base**, per evitare una catena infinita di chiamate.
- Se il caso base non viene mai raggiunto, lesecuzione termina con errore di **stack overflow**.

Consumo di memoria:

- Ogni funzione (ricorsiva o no) richiede memoria per:
  - il codice della funzione;
  - i parametri in ingresso;
  - le variabili locali.

Stack di Sistema nelle Funzioni Ricorsive

Una funzione ricorsiva è composta da due componenti principali:

- **Caso base**: il caso in cui la funzione restituisce un valore senza ulteriori chiamate.
- **Chiamata ricorsiva**: la funzione si richiama per risolvere un sottoproblema, che porterà infine al caso base.

Lo Stack di Sistema: Quando una funzione ricorsiva viene chiamata, il sistema memorizza lo stato della funzione in una struttura chiamata *stack*. Ogni chiamata aggiunge un nuovo *frame* allo stack contenente:

- Valori dei parametri della funzione.
- Posizione nel codice della chiamata.

Funzionamento dello Stack: Quando la funzione raggiunge il caso base, lo stack si svuota rimuovendo i frame in ordine inverso.

Complessità di Spazio nelle Funzioni Ricorsive

La **complessità di spazio** di una funzione ricorsiva è determinata dalla profondità massima dello stack, ovvero il numero massimo di frame attivi nello stack durante l'esecuzione.

Esempio: Calcolo del Fattoriale

```
def fattorialeR(n):
    if n == 0:
        return 1
    return n * fattorialeR(n - 1)
```

Per il calcolo di 5!, lo stack avrà i seguenti frame:

- *fattoriale*(5) $\rightarrow$ invoca *fattoriale*(4)
- *fattoriale*(4) $\rightarrow$ invoca *fattoriale*(3)
- *fattoriale*(3) $\rightarrow$ invoca *fattoriale*(2)
- *fattoriale*(2) $\rightarrow$ invoca *fattoriale*(1)
- *fattoriale*(1) $\rightarrow$ invoca *fattoriale*(0)
- *fattoriale*(0) $\rightarrow$ ritorna 1 (caso base)

La **complessità di spazio** per questa funzione è $O(n)$, poiché ci sono n chiamate ricorsive prima di arrivare al caso base. Ogni chiamata rivorsiva aggiunge un nuovo frame, e quindi la profondità massima dello stack è n.

Ricorsione vs Iterazione: Vantaggi e Svantaggi

Quando preferire la ricorsione?

- Quando il problema ha una struttura **naturalmente ricorsiva**, e la soluzione risulta più chiara.

Quando evitarla?

- Quando esiste una **versione iterativa altrettanto semplice**.
- Quando è richiesta **massima efficienza** in termini di memoria e prestazioni.

Esempio iterativo: fattoriale

```
def fattorialeI(n):
    f = 1
    for i in range(1, n+1):
        f *= i
    return f
```

In questo caso, l'implementazione iterativa è più efficiente e altrettanto leggibile.

Pro e contro della ricorsione:

- **Vantaggi:** Codice più compatto e leggibile.
- **Svantaggi:** Maggiore uso di memoria.

Esempio 1 Somma Ricorsiva di una Lista (1/2)

Obiettivo: progettare una procedura ricorsiva per calcolare la somma degli elementi di una lista di interi.

Idea di base:

- La somma di una lista vuota è 0.
- Altrimenti, è pari al primo elemento più la somma del resto della lista.

Implementazione con slicing:

```
def SommaLista(A):                          # T(n) =
    if len(A) == 0:                         # O(1)
        return 0                            # O(1)
    return A[0] + SommaLista(A[1:])   # O(n) + T(n-1)
```

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 0 \\ T(n-1) + \Theta(n) & \text{altrimenti} \end{cases} \Rightarrow T(n) = \Theta(n^2)$$

Esempio 1 Somma Ricorsiva di una Lista (2/2)

Versione più efficiente (senza slicing):

```
def SommaLista(A, i=0):                    # T(n) =
    if i == len(A):                        # O(1)
        return 0                           # O(1)
    return A[i] + SommaLista(A, i+1)   # O(1) + T(n-1)
```

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 0 \\ T(n-1) + \Theta(1) & \text{altrimenti} \end{cases} \Rightarrow T(n) = \Theta(n)$$

Nota: evitare lo *slicing* `A[1:]` in funzioni ricorsive su liste di grandi dimensioni: ha costo $O(b-a)$ e può accumularsi.
Meglio: iterare sulla lista usando un indice.

Esempio 2 Calcolo Ricorsivo di x^n (1/2)

Obiettivo: progettare una procedura ricorsiva per calcolare x^n, dove x è reale e n è un intero non negativo.

Osservazione: per $n > 0$ si ha:

$$x^n = x \cdot x^{n-1}$$

Codice ricorsivo semplice:

```
def power(x, n):
    # Caso base: qualsiasi numero elevato a 0 e' 1
    if n == 0:
        return 1
    return x * power(x, n - 1)
```

Complessità:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 0 \\ T(n-1) + \Theta(1) & \text{altrimenti} \end{cases} \Rightarrow T(n) = \Theta(n)$$

Esempio 2 Calcolo Ricorsivo di x^n (2/2)

Idea: migliorare la procedura distinguendo i casi pari e dispari.

- **Caso pari:** $x^n = x^{\frac{n}{2}} \cdot x^{\frac{n}{2}}$
- **Caso dispari:** $x^n = x \cdot x^{n-1}$

Codice ottimizzato:

```
def power(x, n):
    if n == 0:
        return 1
    if n % 2 == 0:
        a = power(x, n // 2)
        return a * a
    else:
        a = power(x, (n - 1) // 2)
        return x * a * a
```

Complessità:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 0 \\ T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + \Theta(1) & \text{altrimenti} \end{cases} \Rightarrow T(n) = \Theta(\log n)$$

Esempio 3 Palindromi

Verifica se una lista è palindroma:

Una lista è palindroma se:

- Ha al più un elemento, oppure
- Gli estremi sono uguali e la sottolista centrale è anch'essa palindroma

Esempio: `[1,5,4,8,4,5,1]`

Codice ricorsivo:

```
def palindroma(A, inizio, fine):                    # T(n) =
    if inizio >= fine:                              # O(1)
        return True                                 # O(1)
    if A[inizio] != A[fine]:                        # O(1)
        return False                                # O(1)
    return palindroma(A, inizio+1, fine-1)   # O(1) + T(n-2)
```

Complessità:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n \leq 1 \\ T(n-2) + \Theta(1) & \text{altrimenti} \end{cases} \Rightarrow T(n) = \Theta(n)$$

Esempio 4 - Permutazioni di $\{1, \ldots, n\}$

Genera la lista delle $n!$ permutazioni dei numeri in $\{1, 2, \ldots, n\}$:
Ad esempio, per $n = 3$ si ha:

$$[[3, 2, 1], [2, 3, 1], [2, 1, 3], [3, 1, 2], [1, 3, 2], [1, 2, 3]]$$

Idea: Se $n = 1$, la sola permutazione è $[1]$. Altrimenti:

- Calcoliamo le permutazioni di $\{1, \ldots, n-1\}$
- Inseriamo n in tutte le possibili posizioni di ogni permutazione ottenuta

Pseudocodice:

```
def Permuta(n):
    if n == 1:
        return [[1]]
    else:
        risultato = []
        for p in Permuta(n - 1):
            for i in range(n ):
                nuova = p[:i] + [n] + p[i:]
                risultato.append(nuova)
        return risultato
```

Complessità:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n \leq 1 \\ n \cdot T(n-1) + \Theta(n!) & \text{altrimenti} \end{cases} \Rightarrow T(n) = \Theta(n \cdot n!)$$

Esempio 5 Numeri di Fibonacci (1/4)

Definizione matematica:

$$F(0) = 0, \quad F(1) = 1, \quad F(n) = F(n-1) + F(n-2) \quad \text{per } n > 1$$

Primi valori della successione:

$$0,\ 1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ \ldots$$

Formula chiusa (Binet):

$$F(n) = \frac{\Phi^n - (1-\Phi)^n}{\sqrt{5}} \quad \text{dove } \Phi = \frac{1+\sqrt{5}}{2}$$

Nota: la formula di Binet può introdurre errori numerici con valori grandi di n a causa degli arrotondamenti.

Esempio 5 Numeri di Fibonacci (2/4)

Implementazione ricorsiva diretta:

```
def FibonacciR(n):
    if n <= 1:
        return n
    return FibonacciR(n-1) + FibonacciR(n-2)
```

Ricorrenza della complessità:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n \leq 1 \\ T(n-1) + T(n-2) + \Theta(1) & \text{altrimenti} \end{cases} \Rightarrow T(n) = \Theta(2^n)$$

Problema: La crescita esponenziale è causata da molte chiamate ridondanti. Ad esempio, nel calcolo di $F(5)$, $F(3), F(2), F(1), F(0)$ vengono calcolati più volte.

Esempio 5 Numeri di Fibonacci (3/4)

Versione iterativa :

```
def FibonacciI(n):
    F = [None] * (n + 1)
    F[0], F[1] = 0, 1
    for i in range(2, n + 1):
        F[i] = F[i-1] + F[i-2]
    return F[n]
```

Complessità:

$$\text{Tempo: } \Theta(n), \quad \text{Spazio: } \Theta(n)$$

Vantaggio: ogni valore viene calcolato una sola volta e salvato per riutilizzo.

Esempio 5 Numeri di Fibonacci (4/4)

Versione iterativa ottimizzata in spazio:

```
def FibonacciI(n):
    a, b = 0, 1
    for i in range(2, n + 1):
        a, b = b, a + b
    return b
```

Osservazione: Per calcolare $F(n)$ servono solo $F(n-1)$ e $F(n-2)$. Non è necessario salvare l'intera sequenza.

Complessità:

$$\text{Tempo: } \Theta(n), \quad \text{Spazio: } \Theta(1)$$

Conclusione: Prestazioni ottimali in tempo e spazio con codice semplice.

Esempio 4 Numeri di Fibonacci (5/5)

Confronto sperimentale tra Fibonacci ricorsivo e Fibonacci iterativo

Funzione	Tempo stimato (secondi)
`FibonacciR(20)`	≈ 0.001
`FibonacciR(30)`	≈ 0.1
`FibonacciR(40)`	≈ 10
`FibonacciR(50)`	≈ 33
`FibonacciI(50)`	≈ 0.00001
`FibonacciI(500)`	≈ 0.0001
`FibonacciI(5000)`	≈ 0.001

Tabella: Tempi di calcolo stimati per le funzioni `FibonacciR(n)` e `FibonacciI(n)`. Questi tempi sono approssimativi, dipendono dallhardware, ma danno unidea della differenza di efficienza.

Osservazioni:

- L'algoritmo `FibonacciR(n)` ha complessità esponenziale: il numero di chiamate ricorsive cresce rapidamente, portando a tempi di calcolo insostenibili già per $n = 40$.
- Per $n = 50$, il tempo richiesto supera i 30 secondi, rendendo la versione ricorsiva impraticabile.
- L'algoritmo `FibonacciI(n)` ha complessità lineare $O(n)$: i tempi di calcolo rimangono contenuti anche per valori elevati di n, come $n = 5000$.
- La differenza di efficienza evidenzia l'importanza dell'analisi della complessità algoritmica e della scelta dell'approccio corretto.

ESERCIZI

Suggerimento: Prima di cercare le soluzioni in rete, prova a ragionarci da solo: gli esercizi che seguono sono pensati per aiutarti a mettere alla prova la tua comprensione, la tua capacità di sviluppare una prova di correttezza e costruire un algoritmo corretto.
Solo dopo averci riflettuto, confronta la tua soluzione con altre possibili versioni.

Esercizi

Per ciascuno dei seguenti problemi:

- Progettate in Python una procedura ricorsiva per risolverli, assicurandovi che la complessità sia descritta dalla seguente relazione di ricorrenza:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 0 \\ T(n-1) + \Theta(1) & \text{altrimenti} \end{cases}$$

- Fornite successivamente una soluzione iterativa con complessità $\Theta(n)$.

problemi da risolvere:

1. Data una lista di numeri interi, calcolare la somma di tutti i numeri pari presenti nella lista. Ad esempio per $lista = [3, 7, -6, 2, 3, -1]$ la risposta è -4.
2. Data una lista non vuota di numeri, determinare il valore minimo e il valore massimo nella lista. Da esempio per $lista = [3, 7, -6, 2, 3, -1]$ la risposta è $(-6, 7)$.
3. Date due stringhe binarie della stessa lunghezza, determinare il numero di posizioni in cui differiscono.
   Ad esempio per $s_1 = "1101"$ ed $s_2 = "1001"$ la risposta è 2.
4. Stampare gli elementi di una lista in ordine inverso. Ad esempio per la lista $[1, 2, 3]$, l'output dovrebbe essere $3, 2, 1$.

Esercizi

1. Scrivete una procedura ricorsiva in Python per convertire un numero decimale n in una stringa che rappresenti il suo equivalente in binario.
   Ad esempio, per $n = 23$, la funzione dovrebbe restituire la stringa "10111"
   La complessità della soluzione proposta deve essere descritta dalla seguente equazione di ricorrenza: $T(n) = T(\lfloor \frac{n}{2} \rfloor) + \Theta(\log n)$.
2. Scrivete una procedura ricorsiva in Python che, data una lista A di numeri interi, restituisca il numero di coppie (i, j) tali che $A[i] = A[j]$ e $i < j$).
   Ad esempio, per la lista $[1, 1, 2, 1, 2]$, la risposta dovrebbe essere 4 (le coppie sono $(0, 1)$, $(0, 3)$, $(1, 3)$, $(2, 4)$).
   La complessità della soluzione proposta deve essere descritta dalla seguente equazione di ricorrenza: $T(n) = T(n-1) + \Theta(n)$.
3. Scrivete una funzione ricorsiva in Python che, data una lista di n numeri interi, restituisca il numero di sottoliste (sequenze contigue di elementi) in cui la somma degli elementi è pari.
   Ad esempio, per la lista $A = [1, 2, 3, 4]$, la risposta dovrebbe essere 4 (le sottoliste con somma pari sono: $([2],\ [4],\ [1, 2, 3],\ [1, 2, 3, 4])$.
   La complessità della soluzione proposta deve essere descritta dalla seguente equazione di ricorrenza: $T(n) = T(n-1) + \Theta(n)$.
4. Scrivete una procedura ricorsiva in Python che, data una lista di n numeri interi e un numero intero x, determini se esiste un sottoinsieme degli elementi della lista la cui somma sia esattamente x.
   Ad esempio, per la lista $[3, -34, 4, -12, 5, 2]$ e $x = -28$ la risposta dovrebbe essere *True*, (un sottoinsieme che somma a -28 è $\{-34, 4, 2\}$)
   La complessità della soluzione proposta deve essere descritta dalla seguente equazione di ricorrenza: $T(n) = 2T(n-1) + O(1)$.