Definizione Formale

La ricorsione è una tecnica di programmazione che permette di risolvere un problema scomponendolo in sottoproblemi analoghi di minore dimensione. Un metodo ricorsivo è definito in termini di se stesso, invocandosi direttamente o indirettamente.

Struttura di un Algoritmo Ricorsivo

Un algoritmo ricorsivo corretto è composto da:

1. **Caso base (condizione di chiusura)**: definisce la soluzione per problemi di dimensione minima, risolvibili direttamente
2. **Passo induttivo (parte ricorsiva)**: definisce come risolvere il problema originale utilizzando la soluzione di uno o più sottoproblemi di dimensione inferiore

Tipologie di Ricorsione

Ricorsione Diretta

Il metodo chiama direttamente se stesso.

Ricorsione Indiretta

Un metodo A chiama un metodo B che a sua volta chiama il metodo A, direttamente o tramite ulteriori chiamate intermedie.

Ricorsione Multipla

Un metodo invoca più volte se stesso in una singola esecuzione (es. algoritmo di Fibonacci).

Ricorsione Infinita

Si verifica quando:

- I valori dei parametri non si semplificano verso il caso base
- Manca la clausola di chiusura
- La condizione di terminazione è errata

Analisi del Funzionamento

Stack di Attivazione

Ogni chiamata ricorsiva comporta:

- Allocazione di un nuovo frame nello stack
- Creazione di nuove istanze di variabili locali e parametri
- Memorizzazione dell'indirizzo di ritorno
- Salvataggio dello stato di esecuzione

Traccia di Esecuzione

L'esecuzione di un metodo ricorsivo procede in due fasi:

1. **Fase discendente**: dal problema originale ai casi base, accumulando chiamate nello stack
2. **Fase ascendente**: risalita dallo stack, calcolando le soluzioni dai casi base fino al problema originale

Equivalenza con Iterazione

- Ogni algoritmo ricorsivo può essere trasformato in un algoritmo iterativo equivalente
- Ogni algoritmo iterativo può essere espresso in forma ricorsiva
- La trasformazione da ricorsivo a iterativo può richiedere l'uso esplicito di una struttura dati stack

Vantaggi e Limitazioni

Vantaggi

- Eleganza e compattezza del codice
- Maggiore leggibilità per problemi intrinsecamente ricorsivi
- Semplicità concettuale per alcuni algoritmi (es. attraversamento di strutture dati gerarchiche)

Limitazioni

- Overhead computazionale dovuto alla gestione dello stack
- Maggiore consumo di memoria
- Rischio di stack overflow per ricorsioni profonde
- Minore efficienza rispetto alle controparti iterative

Criteri di Utilizzo Ottimale

La ricorsione è preferibile quando:

- L'efficienza non è un fattore critico
- La natura del problema è intrinsecamente ricorsiva
- La soluzione iterativa è complessa o poco intuitiva

- Il problema si scompone naturalmente in sottoproblemi identici

Algoritmi Ricorsivi Paradigmatici

Fattoriale

```
static int fattoriale(int n) {
    if (n == 0)
        return 1;  // caso base
    else
        return n * fattoriale(n-1);  // passo induttivo
}
```

Ricerca in Strutture Ricorsive

```
static boolean cerca(Nodo radice, Elemento e) {
    if (radice == null)
        return false;  // caso base: elemento non trovato
    if (radice.valore.equals(e))
        return true;   // caso base: elemento trovato

    // passo induttivo: cerca nei sottoalberi
    return cerca(radice.sinistro, e) || cerca(radice.destro, e);
}
```

Ordinamento Ricorsivo (QuickSort)

```
static void quickSort(int[] array, int inizio, int fine) {
    if (inizio < fine) {
        int indicePartizione = partiziona(array, inizio, fine);

        // ricorsione sulle due partizioni
        quickSort(array, inizio, indicePartizione - 1);
        quickSort(array, indicePartizione + 1, fine);
    }
}
```

Ottimizzazioni

Tail Recursion (Ricorsione in coda)

Si verifica quando la chiamata ricorsiva è l'ultima operazione del metodo. Può essere ottimizzata dal compilatore eliminando l'overhead dello stack.

Memoizzazione

Tecnica che memorizza i risultati già calcolati per evitare ricalcoli in algoritmi con sottoproblemi sovrapposti (es. Fibonacci).

Considerazioni Implementative

1. Garantire la convergenza verso il caso base
2. Verificare la correttezza delle condizioni di terminazione
3. Limitare la profondità ricorsiva in base ai vincoli hardware
4. Considerare alternative iterative per algoritmi a elevata complessità computazionale