📄 Pseudocodice Completo Commentato

Questo file raccoglie **tutti** i frammenti di pseudocodice del progetto, con commenti esplicativi per ciascuna funzione.

---

1. crea()

```
// Crea un nuovo dizionario vuoto
funzione crea() → Dizionario
    diz ← nuovo Dizionario
    diz.parole ← mappa vuota      // inizializza insieme delle parole
    diz.schemi ← mappa vuota      // inizializza insieme degli schemi
    ritorna diz
```

---

2. Inserisci

```
// Aggiunge una parola o uno schema al dizionario
funzione Inserisci(d: *Dizionario, w: string)
    se w contiene almeno una maiuscola allora
        se w non in d.schemi allora
            aggiungi w a d.schemi  // inserisce schema
    altrimenti
        se w non in d.parole allora
            aggiungi w a d.parole  // inserisce parola
```

---

3. Elimina

```
// Rimuove una parola o uno schema dal dizionario
funzione Elimina(d: *Dizionario, w: string)
    se w in d.parole allora
        rimuovi w da d.parole     // elimina parola
    se w in d.schemi allora
        rimuovi w da d.schemi     // elimina schema
```

---

4. Carica

```
// Carica parole e schemi da un file di testo
funzione Carica(d: *Dizionario, percorso: string)
    apre file in lettura
    per ogni token w nel file (separato da spazi o newline):
        chiama Inserisci(d, w)    // riutilizza Inserisci per gestire
duplicati
    chiudi file
```

5. StampaParole e StampaSchemi

```
// Stampa tutte le parole racchiuse tra parentesi quadre
funzione StampaParole(d: *Dizionario)
    stampa "["
    per ogni w in d.parole:
        stampa w
    stampa "]"

// Stampa tutti gli schemi racchiusi tra parentesi quadre
funzione StampaSchemi(d: *Dizionario)
    stampa "["
    per ogni S in d.schemi:
        stampa S
    stampa "]"
```

6. Compatibile (schema ↔ parola)

```
// Verifica se una parola è compatibile con uno schema
funzione Compatibile(S: string, w: string) → boolean
    se len(S) ≠ len(w) allora
        ritorna False            // lunghezza diversa: non
compatibili
    mappaMaiuscole ← dizionario vuoto
    per i da 0 a len(S)-1:
        a ← S[i]                // carattere dello schema
        b ← w[i]                // carattere della parola
        se a è maiuscola allora
            se mappaMaiuscole[a] non esiste allora
                mappaMaiuscole[a] ← b
            altrimenti se mappaMaiuscole[a] ≠ b allora
                ritorna False    // vincolo di mapping violato
        altrimenti              // a è minuscola
            se a ≠ b allora
                ritorna False    // lettera minuscola non combacia
    ritorna True                 // tutte le condizioni soddisfatte
```

7. Damerau-Levenshtein (distanza di editing)

```
// Calcola il numero minimo di operazioni di editing (inserzione,
cancellazione,
// sostituzione, scambio) per trasformare x in y
funzione DamerauLevenshtein(x: string, y: string) → integer
    n ← len(x), m ← len(y)
    D ← matrice di interi dimensione (n+1)×(m+1)

    // inizializzazione delle basi
    per i in 0..n:
        D[i][0] ← i
    per j in 0..m:
        D[0][j] ← j

    // calcolo dinamico
    per i in 1..n:
        per j in 1..m:
            costo ← (x[i-1] == y[j-1]) ? 0 : 1
            D[i][j] ← min(
                D[i-1][j] + 1,          // cancellazione
                D[i][j-1] + 1,          // inserzione
                D[i-1][j-1] + costo     // sostituzione
            )
            // controllo scambio caratteri adiacenti
            se i > 1 e j > 1 e x[i-1] == y[j-2] e x[i-2] == y[j-1]
allora
                D[i][j] ← min(D[i][j], D[i-2][j-2] + 1)
    ritorna D[n][m]
```

8. Catena (path minimo con distanza = 1)

```
// Trova la sequenza minima di parole con distanza di editing 1
funzione Catena(d: *Dizionario, x: string, y: string) →
elenco<string> o nil
    se x non in d.parole o y non in d.parole allora
        ritorna nil             // parole non presenti
    visitati ← insieme{x}
    queue ← coda inizializzata con [[x]]  // ogni elemento è un
percorso

    mentre queue non vuota:
        path ← queue.pop_front()
        last ← path[-1]        // ultima parola del percorso
        se last == y allora
            ritorna path       // trovato percorso minimo
        per ogni w in d.parole:
            se w non in visitati e DamerauLevenshtein(last, w) == 1
allora
                aggiungi w a visitati
                queue.push(path + [w])
    ritorna nil                // nessun percorso trovato
```

9. Gruppo (componente connessa di parole)

```
// Trova tutte le parole raggiungibili da x tramite distanze di
editing 1
funzione Gruppo(d: *Dizionario, x: string) → elenco<string> o nil
    se x non in d.parole allora
        ritorna nil
    visitati ← insieme{x}
    queue ← [x]
    result ← [x]

    mentre queue non vuota:
        curr ← queue.pop_front()
        per ogni w in d.parole:
            se w non in visitati e DamerauLevenshtein(curr, w) == 1
allora
                aggiungi w a visitati
                queue.push(w)
                result.append(w)
    ritorna result
```

10. CostruisciGrafoSchemi

```
// Crea un grafo in cui due schemi sono connessi se esiste almeno una
parola compatibile
funzione CostruisciGrafoSchemi(schemi: insieme<string>, parole:
insieme<string>) → dizionario<schema, lista<schema>>
    grafo ← dizionario vuoto
    per ogni A in schemi:
        per ogni B in schemi:
            se A ≠ B allora
                per ogni w in parole:
                    se Compatibile(A, w) e Compatibile(B, w) allora
                        grafo[A].append(B)
                        esci loop parole
    ritorna grafo
```

11. Famiglia (componente connessa di schemi)

```
// Trova la famiglia di schemi connessi tramite compatibilità
funzione Famiglia(S: string, d: *Dizionario) → elenco<string> o nil
    se S non in d.schemi allora
        ritorna nil
    grafo ← CostruisciGrafoSchemi(d.schemi, d.parole)
    visitati ← insieme{S}
    queue ← [S]
    result ← [S]

    mentre queue non vuota:
        curr ← queue.pop_front()
        per ogni neigh in grafo[curr]:
            se neigh non in visitati allora
                aggiungi neigh a visitati
                queue.push(neigh)
                result.append(neigh)
    ritorna result
```

12. esegui (gestione dei comandi)

```
// Interpreta ed esegue i comandi letti da input
funzione esegui(d: *Dizionario, riga: string)
    token ← split(riga)  // campi separati da spazi

    se token vuoto: ritorna

    op ← token[0]
    switch op:

        caso "c" se len(token) == 1:
            // reset completo del dizionario
            d = crea()          // oppure *d = *crea()
```

```
        d = crea()          // oppure *d = *crea()

    caso "c" se len(token) == 2:
        Carica(d, token[1])

    caso "p":
        StampaParole(d)

    caso "s":
        StampaSchemi(d)

    caso "i":
        Inserisci(d, token[1])

    caso "e":
        Elimina(d, token[1])

    caso "r":
        stampa token[1] + ":["
        per ogni w in d.parole:
            se Compatibile(token[1], w) allora
                stampa w
        stampa "]"

    caso "d":
        stampa DamerauLevenshtein(token[1], token[2])

    caso "c":
        path ← Catena(d, token[1], token[2])
        se path == nil allora
            stampa "non esiste"
        altrimenti
            stampa "("
            per ogni w in path:
                stampa w
            stampa ")"

    caso "g":
        grp ← Gruppo(d, token[1])
        se grp == nil allora
            stampa "non esiste"
        altrimenti
            stampa "[" + grp + "]"

    caso "f":
        fam ← Famiglia(token[1], d)
        se fam == nil allora
            stampa "non esiste"
        altrimenti
            stampa "[" + fam + "]"

    default:
        // comando non riconosciuto → ignora o segnala errore
```