Esercizio 2 - edit_distance_dyn

In questo esercizio è stata implementata una versione ricorsiva dell'algoritmo edit_distance che utilizza la strategia della programmazione dinamica.
Prima di arrivare alla "versione finale" dell'esercizio abbiamo svolto vari esperimenti:
Tutti gli esperimenti sono stati effettuati con i file di input forniti a lezione.

- **Base**
  Algoritmo ricorsivo il più vicino possibile all'algoritmo visto a lezione.
  Questa versione risulta essere molto lenta a causa degl'alberi di ricorsione che calcolano più volta la distanza tra due sottostringhe uguali. Più una parola è lunga più il tempo d'esecuzione cresce arrivando ad ore di esecuzione
- **Memorizzazione mediante albero rosso-nero**
  Io ed il mio collega abbiamo pensato che forse era possibile ridurre il numero di chiamate alla funzione edit_distance memorizzando in un albero biancicato tutti i confronti effettuati da una parola con tutto il dizionario. L'albero rosso avrebbe permesso un tempo di ricerca nella memoria di complessità O(log(N)), riducendo il numero di confronti calcolati dalla funzione dato che in memoria sarebbero stati presenti molte più combinazioni.
  Purtroppo quest'idea non è risultata vincente, il tempo d'esecuzione, ovviamente migliore della versione base, è stato superiore a 10 minuti. Il tempo guadagnato per non dover rifare il calcolo tra due stringhe trovate in memoria non ha giustificato il di ricerca nella memoria che anche se pur logaritmico è stato un "colo di bottiglia".
- **Memorizzazione tramite Hash-Table**
  L'idea alla base dello sviluppo di un hash table per la memorizzazione delle combinazioni di stringhe è la stessa del punto precedente: Non calcolare nuovamente la distanza tra due stringhe che ho già incontrato in una parola precedente.
  Per funzionare questa versione avrebbe bisogno di una funzione iniettiva che ad ogni coppia di stringhe associa una posizione univoca. Purtroppo non siamo stati in grado di trovarla/generarla e siamo andati a gestire le collisioni concatenandole.
  Questa soluzione è risultata essere più efficiente dell'albero RN, con un tempo di esecuzione intorno agl'8 minuti. "Il colo di bottiglia" è risultato essere la ricerca del corretto elemento all'interno delle collisioni.
- **Memorizzazione tramite matrice di interi**
  La soluzione che è risultata vincente è stata utilizzare una matrice bidimensionale di interi.
  Per ogni coppia di parole la lunghezza della prima fornisce il primo indice, e la lunghezza della seconda parola il secondo. Utilizzando una matrice vuota all'inizio dell'esecuzione della funzione il tempo di ricerca di una coppia di elementi in memoria è O(1).
  Questo ha permesso di abbassare il tempo di esecuzione a circa 2 minuti.
- **Finale**
  Per evitare di fare operazioni già effettuate in precedenza abbiamo fatto due modifiche alla versione dell'edit_distance vista a lenizione:
  a. Abbiamo aggiunto il parametro MIN contenente la distanza minima già trovata dalla parola che si vuole "correggere". Avendo a disposizione questa variabile abbiamo modificato l'algoritmo per non interrompere le chiamate ricorsive nel caso in cui la distanza tra due stringhe diventi maggiore o uguale alla minima già calcolata.
  b. Abbiamo aggiunto i parametri lenght_str1 lenght_str2 che contengono la dimensione della due stringhe, evitando così di doverle calcolare ad ogni chiamata.