**Braille Auto-Correct**

This document details a streamlined approach to auto-correct Braille input using only the Levenshtein distance for matching, along with optimizations and trade-offs.

**1. Objective**

Provide a simple yet effective real-time suggestion system for QWERTY-based Braille input that:

* Corrects substitutions, insertions, and deletions.
* Requires minimal setup and dependencies.
* Delivers low latency for small-to-medium dictionaries.

**2. Approach**

1. Input Normalization
   * Map keys s, d, f, j, k, l → dots 1–6.
   * Sort and join into a string pattern ("sdf" → "123").
2. Dictionary Preprocessing
   * Precompute pattern for each word in the dictionary once at startup.
3. Brute-Force Matching
   * For each query, iterate all dictionary patterns:
     + Compute Levenshtein distance with early exit.
     + Collect words whose distance ≤ max\_dist.
4. Ranking & Suggestion
   * Sort by (distance, word) to resolve ties.
   * Return the top-K candidates (e.g., 3).

**3. Core Algorithm Components**

3.1. QWERTY-to-Pattern Conversion

* Mapping Dict: {'s':1,'d':2,'f':3,'j':4,'k':5,'l':6}
* Function: Filter valid keys, map, sort, join.

3.2. Levenshtein Distance

* DP Table: Compute in O(n·m) time, where *n*, *m* are string lengths.
* Space: O(min(n,m)) by storing only the previous row.
* Early Exit: Abort when the current row’s minimal edit cost exceeds max\_dist.

**4. Optimizations**

* Early-Abandon: Greatly reduces work on dissimilar patterns by cutting DP loops early.
* One-Time Preprocessing: Patterns computed just once, reducing per-query overhead.
* Compact Patterns: Using short digit strings speeds up string comparisons and distance calls.
* Threshold Tuning: Adjusting max\_dist to balance recall vs. computation (e.g., 1–2 for typical Braille patterns).

**5. Trade-Offs Analysis**

| Choice | Benefit | Drawback |
| --- | --- | --- |
| Brute-Force Search | Simplicity; no additional data structures | O(N·n·m) per query may be slow for large N |
| Levenshtein | Handles all typo types uniformly | Quadratic with pattern length |
| Pure Python | Portable; no external deps | Single-threaded performance ceiling |
| Early-Termination | Cuts off costly comparisons early | Requires careful threshold setting |

**6. Suitability & Extensions**

* When to Use: Prototyping, educational demos, dictionaries up to ~50K entries.
* Scalability: For larger datasets or microsecond latency, layer in an index (e.g., BK-tree) or compile hotspots in C/C++.
* Future Enhancements:
  + Adopt Damerau–Levenshtein for transposition sensitivity.
  + Parallelize distance computations across threads or processes.
  + Integrate approximate hashing (LSH) for constant-time lookup.