**Student:** Hong Quan Doan

**Student ID:** 986956

**Day 14, 15 – Mar 12, 13**

**Environment**

MacBook, Core i5, 4 CPU, 8GB

**Best Matching Words**

**Test results:**

\*\*\*\*\*\* BestMatching Serial \*\*\*\*\*\*

Dictionary Size: 250353

Word: stitter

Minimun distance: 1

List of best matching words: 9

sitter

skitter

slitter

spitter

stilter

stinter

stotter

stutter

titter

Execution Time: 363.0

\*\*\*\*\*\* BestMatching Basic Concurrent \*\*\*\*\*\*

Dictionary Size: 250353

Word: stitter

Minimun distance: 1

List of best matching words: 9

sitter

skitter

slitter

spitter

stilter

stinter

stotter

stutter

titter

Execution Time: 209.0

Speedup: 1.736842

\*\*\*\*\*\* BestMatching Advanced Concurrent \*\*\*\*\*\*

Dictionary Size: 250353

Word: stitter

Minimun distance: 1

List of best matching words: 9

sitter

skitter

slitter

spitter

stilter

stinter

stotter

stutter

titter

Execution Time: 172.0

Speedup: 2.110465

**Source code:**

For BasicConcurrent version of lamda

**public** **static** BestMatchingData getBestMatchingWords(String word, List<String> dictionary) **throws** InterruptedException, ExecutionException {

**int** numCores = Runtime.*getRuntime*().availableProcessors();

ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.*newFixedThreadPool*(numCores);

**int** size = dictionary.size();

**int** step = size / numCores;

**int** remainder = size % numCores;

List<Future<BestMatchingData>> results = **new** ArrayList<>();

**for** (**int** i = 0; i < numCores; i++) {

**final** **int** startIndex = i \* step;

**final** **int** endIndex = (i \* step) + step + ((i == numCores - 1) ? remainder : 0);

Callable<BestMatchingData> bmd = () -> {

List<String> res = **new** ArrayList<String>();

**int** minDistance = Integer.***MAX\_VALUE***;

**int** distance;

**for** (**int** j = startIndex; j < endIndex; j++) {

distance = LevenshteinDistance.*calculate*(word, dictionary.get(j));

**if** (distance < minDistance) {

res.clear();

minDistance = distance;

res.add(dictionary.get(j));

} **else** **if** (distance == minDistance) {

res.add(dictionary.get(j));

}

}

BestMatchingData result = **new** BestMatchingData();

result.setWords(res);

result.setDistance(minDistance);

**return** result;

};

Future<BestMatchingData> future = executor.submit(bmd);

results.add(future);

}

executor.shutdown();

List<String> words = **new** ArrayList<String>();

**int** minDistance = Integer.***MAX\_VALUE***;

**for** (Future<BestMatchingData> future : results) {

BestMatchingData data = future.get();

**if** (data.getDistance() < minDistance) {

words.clear();

minDistance = data.getDistance();

words.addAll(data.getWords());

} **else** **if** (data.getDistance() == minDistance) {

words.addAll(data.getWords());

}

}

BestMatchingData result = **new** BestMatchingData();

result.setDistance(minDistance);

result.setWords(words);

**return** result;

}

For AdvancedConcurrent version of lamda

List<Callable<BestMatchingData>> tasks = **new** ArrayList<>();

**for** (**int** i = 0; i < numCores; i++) {

**final** **int** startIndex = i \* step;

**final** **int** endIndex = (i \* step) + step + ((i == numCores - 1) ? remainder : 0);

Callable<BestMatchingData> bmd = () -> {

List<String> res = **new** ArrayList<String>();

**int** minDistance = Integer.***MAX\_VALUE***;

**int** distance;

**for** (**int** j = startIndex; j < endIndex; j++) {

distance = LevenshteinDistance.*calculate*(word, dictionary.get(j));

**if** (distance < minDistance) {

res.clear();

minDistance = distance;

res.add(dictionary.get(j));

} **else** **if** (distance == minDistance) {

res.add(dictionary.get(j));

}

}

BestMatchingData result = **new** BestMatchingData();

result.setWords(res);

result.setDistance(minDistance);

**return** result;

};

tasks.add(bmd);

}

results = executor.invokeAll(tasks);

executor.shutdown();