2 Introduction to the Standard Library

Objective: to solve a moderately large problem using C++. Some parts of the standard library that haven't yet been introduced in the course will be used. Information about these is given in section 3.

1 Spelling Correction

Most word processors can check the spelling of a document and suggest corrections to misspelled words. Often, a dictionary is used — words that aren't in the dictionary are considered to be misspelled. The suggested corrections are the words in the dictionary that are "similar" to the misspelled word.

Your task is to write a class Dictionary which can be used as in the following example:

```
int main() {
    Dictionary dict;
    string word;
    while (cin >> word) {
        transform(word.begin(), word.end(), word.begin(), ::tolower);
        if (dict.contains(word)) {
            cout << "Correct." << endl;
        } else {
            vector<string> suggestions = dict.get_suggestions(word);
            if (suggestions.empty()) {
                cout << "Wrong, no suggestions." << endl;
            } else {
                cout << "Wrong. Suggestions:" << endl;
                for (const auto& w : suggestions) {
                    cout << "    " << w << endl;
                }
            }
        }
    }
}
```

Examples:

```
expertise
Correct.
seperate
Wrong. Suggestions:
    separate
    desperate
    federate
    generate
    imperate
```

Notes:

- The function contains (section 2.2) must be efficient (fast).
- In get_suggestions you can spend time on finding good suggestions for corrections.
- It can be advantageous to "preprocess" the file which contains the dictionary (section 2.1).
- It is not certain that the data structures which you shall use are optimal (or even necessary), but you shall solve the assignments as they are given. You are encouraged to improve the program, but do that as a separate project.

The following shall be done in `get_suggestions`:

1. Search the dictionary and find candidates for corrections (section 2.3). To begin with, the words in the dictionary which have approximately the same number of letters (plus/minus one letter) as the misspelled word should be considered. Of these candidates, the words which contain at least half of the "trigrams" of the misspelled word should be kept. A trigram is three adjacent letters — for example, the word `summer` contains the trigrams `sum umm mme mer`.
2. Sort the candidate list so the "best" candidates are first in the list (section 2.4). The sort key is the cost to change the misspelled word to one of the candidate words.
3. Keep the first 5 candidates in the list (section 2.5).

Expressed in program code:

```
vector<string> Dictionary::get_suggestions(const string& word) const {
    vector<string> suggestions;
    add_trigram_suggestions(suggestions, word);
    rank_suggestions(suggestions, word);
    trim_suggestions(suggestions);
    return suggestions;
}
```

2 Assignments

2.1 Preprocess the Dictionary

A1. The file */usr/share/dict/words* contains a large number of words (one word per line). The file is UTF-8 encoded; ignore this and treat all characters as 8-bit. Write a program which reads the file and creates a new file *words.txt* in the current directory. Each line in the file shall contain a word, the number of trigrams in the word, and the trigrams.[6] The trigrams shall be sorted in alphabetical order; upper case letters shall be changed to lower case. Example:

```
...
hand 2 and han
handbag 5 and bag dba han ndb
handbook 6 and boo dbo han ndb ook
...
```

Copy the *Makefile* from the *lab1* directory, modify it to build the program, build, test.

2.2 Determine If a Word is Correct

A2. Implement the constructor and the function `contains` in `Dictionary`. The preprocessed list of words is in the file *words.txt*. The words shall be stored in an `unordered_set<string>`. Wait with the trigrams until assignment A4.

Modify the makefile (the main program shown in section 1 is in *spell.cc*), build, test.

2.3 Use Trigrams to Find Candidates

A3. The words together with their trigrams must be stored in the dictionary. Each word shall be stored in an object of the following class:

[6] Note that there are short words with zero trigrams.

```
class Word {
public:
    /* Creates a word w with the sorted trigrams t */
    Word(const std::string& w, const std::vector<std::string>& t);

    /* Returns the word */
    std::string get_word() const;

    /* Returns how many of the trigrams in t that are present
       in this word's trigram vector */
    unsigned int get_matches(const std::vector<std::string>& t) const;
};
```

Implement this class. The trigram vector given to the constructor is sorted in alphabetical order (see assignment A1). The function `get_matches` counts how many of the trigrams in the parameter vector that are present in the word's trigram vector.[7] You may assume that the trigrams in the parameter vector also are sorted in alphabetical order. Use this fact to write an efficient implementation of `get_matches`.

A4. The class `Dictionary` shall have a member variable that contains all words with their trigrams. It must be possible to quickly find words which have approximately the same length as the misspelled word. Therefore, the words shall be stored in the following array:

```
vector<Word> words[25]; // words[i] = the words with i letters,
                        // ignore words longer than 25 letters
```

Modify the `Dictionary` constructor so the `Word` objects are created and stored in `words`, implement the function `add_trigram_suggestions`. Use a constant instead of the number 25.

2.4 Sort the Candidate List

After `add_trigram_suggestions` the suggestion list can contain a large number of candidate words. Some of the candidates are "better" than others. The list shall be sorted so the best candidates appear first. The sorting condition shall be the "edit distance" (also called "Levenshtein distance") from the misspelled word to the candidate word.

The cost $d(i,j)$ to change the i first characters in a word p to the j first characters in another word q can be computed with the following formula (i and j start from 1):

$$d(i,0) = i$$

$$d(0,j) = j$$

$$d(i,j) = \text{minimum of} \begin{cases} \text{if } p_i = q_j \text{ then } d(i-1,j-1) \text{ else } d(i-1,j-1)+1, \\ d(i-1,j)+1, \\ d(i,j-1)+1. \end{cases}$$

The minimum computation considers the cost for replacing a character, inserting a character and deleting a character. The cost to change p to q, that is the edit distance, is $d(p.length, q.length)$.

A5. Implement the function `rank_suggestions`. Do *not* write a recursive function, it would be very inefficient. Instead, let d be a matrix (with $d(i,0) = i$ and $d(0,j) = j$) and compute the elements in row order (dynamic programming). Declare d with the type `int[26][26]` to avoid problems with a dynamically allocated matrix.

[7] You don't have to consider multiple occurrences of the same trigram.

2.5 Keep the Best Candidates

A6. Implement the function `trim_suggestions`.

3 More Information About the Assignments

- In the main program in *spell.cc*, the call to `transform` applies the function `tolower` to all the characters in a string (between `begin()` and `end()`), and stores the function result in the same place. `tolower` converts a character from upper case to lower case. The scope operator `::` is necessary to get the right version of the overloaded `tolower` function.
- To sort a vector `v`, call `sort(v.begin(), v.end())`.
- The standard library class `unordered_set` is in header `<unordered_set>`. An element is inserted in a set with the function `insert(element)`. The function `count(element)` returns the number of occurrences of an element (0 or 1).
- Here's one way to sort the suggested candidates in edit distance order (another way is to use a `map`):
  - Define a `vector` with elements that are pairs with the first component an `int`, the second a `string`: `vector<pair<int, string>>`.
  - Compute the edit distance for each candidate word, insert the distance and the word in the vector: `push_back(make_pair(dist, word))`.
  - Sort the vector (pairs have an `operator<` that first compares the first component, so the elements will be sorted according to increasing edit distance).
  - For a pair `p`, `p.first` is the first component, `p.second` the second component.
- Read more about computing edit distance on the web. You may also consider using the Damerau–Levenshtein distance.
- A vector can be resized to size `n` with `resize(n)`.