STL for Competitive Programming and Software Development
Coding Test - April 2016

Victor Ciura - Technical Lead, Advanced Installer
Gabriel Diaconiţa - Senior Software Developer, Advanced Installer

Intro

Google Autocomplete

As you type in the browser search box, you can find information quickly by seeing search predictions that might be similar to the search terms you're typing.
The suggestions that Google offers all come from how people actually search. For example, type in the word "**cruise**" and you get suggestions like:

```
-----------------------------------------------
Keyword: cruise
Suggested searches for keyword: "cruise"
   -> cruise line
   -> cruise ship
   -> carnival cruise
   -> caribbean cruise
   -> princess cruise
   -> disney cruise
   -> celebrity cruise
   -> norwegian cruise
   -> alaska cruise
   -> ship cruise
-----------------------------------------------
```

These are all real searches that have been done by other people. **Popularity** is a factor in what Google shows. If lots of people who start typing in "**cruise**" then go onto type "**cruise line**" that can help make "**cruise line**" appear as a suggestion.

The Problem

You are given a keywords "database" in the form of a large text file (**keywords.db** - 136MB) containing **search terms (phrases)** used by people in the past (consider this an active search cache). Here is a small *fragment* from this text file:

```
------------------ keywords.db ----------------------
philips lcd 15
15 lcd cheap monitor
cheap 15 lcd monitor
dell e153fp 15 lcd midnight grey 36
lcd tv 15
samsung lcd 15
sony 15 lcd monitor
15 dvd lcd tv
15 inch lcd plasma monitors
-----------------------------------------------------------
```

You may assume that the text file contains only ASCII ***alphanumeric*** characters (English words). Keywords are separated by *space* or *CR/LF*.

The keywords database file is to be considered an **immutable** (read-only) snapshot of the current query **cache** to be used for **your own auto-suggestion engine for search terms**.

Each **line** in the file represents a search phrase used in the past. **For simplicity**, you should **ignore** the fact that the past search terms are organised on separate lines and consider the whole "database" as a *continuous chain of keywords*. A keyword is a sequence of non-whitespace characters (no normalization or input sanitization is needed).

Search phrase:

For simplicity, we shall define a search phrase as a pair of just **two consecutive keywords** in the query database. E.g. *"cruise line", "dell e153fp", "cruise ship", "samsung lcd", "norwegian cruise", "lcd cheap", "sony 15", "cheap monitor"*.

➽ Your first task is to load and **rank** the keywords database. That means ordering all search phrases (as defined above) according with their frequency in the cache (database).
Your program should be able to print to a file the **Top 1000** search phrases with their respective ranks (***occurrence*** frequency).
E.g. Top 10 search phrases from **`keywords.db`** are:

```
-------------------------------------------------------
real estate # 43298
for sale # 38022
new york # 27302
how to # 25068
web site # 21073
las vegas # 19039
cell phone # 17657
of the # 15012
credit card # 14278
web hosting # 11037
-------------------------------------------------------
```

➽ Your second task is to implement your own **auto-suggestion engine** for **10** related searches, based on top search phrases containing the input keyword.
See previous example with suggested searches for keyword: "*cruise*".
This operation (user inputs a new keyword and the engine auto-suggests related search phrases) should be repetitive during a program session and should be **super-fast**.
*** This interactive mode should be active only when the program receives a **`/search`** command-line switch.

➽ Provide an analysis of the **runtime** (steps) and **space** (memory usage) complexity in **Big-O** notation for all functions/code-blocks. The analysis should include the **average** and **worst-case** complexity along with a **brief explanation** of your reasoning. Write this information in **comments**; start them with: **// [SPACE COMPLEXITY]** or **// [RUNTIME COMPLEXITY]**

Official Solution

Check program correctness:

➳ Keyword database **ranking** must be checked by comparing (DIFF) student provided **Top1000.txt** output file with our reference implementation.
The program loads the database and produces the correct ranking in a reasonably fast time. (depending on HW configuration: <1min)

➳ Implemented **auto-suggestion engine** for **10** related searches will be checked by random keyword input and comparing the output with our reference implementation.
The program should output the search suggestions super-fast.

Check the stats:

Input: "keywords.db" - 136 MB (143,157,546 bytes)

- unique keywords = 292,191
- keyword tuples = 4,967,228

Criteria for code evaluation:

- code must compile (without warnings)
- code is succinct and easy to understand
- code uses STL
- code is well commented
- implementation is efficient/performant (CPU utilization, memory utilization)
- the code contains (in comments) an analysis of the **runtime** (steps) and **space** (memory usage) complexity in **Big-O** notation for all functions/code blocks
- data structures used by the algorithm are designed to be space efficient to be able to scale to a much bigger keyword database

Solution 1

Data structures

Data structures used by the algorithm are designed to store the minimal amount of information in memory (no redundancy, no keyword copies).

Data structures leverage STL container iterators that are stable (valid) under the used algorithm operations.

```
// stores unique keywords from input source
using KeywordSet = set<string>;

// a search phrase is composed of a pair/tuple of keywords (from our set)
struct KeywordTuple
{
  KeywordSet::const_iterator firstWord;
  KeywordSet::const_iterator secondWord;

  // BinaryPredicate for sorting keyword tuples lexicographically
  friend bool operator< (const KeywordTuple & kt1, const KeywordTuple & kt2)
  {
    // compare by first keyword
    if ( *kt1.firstWord < *kt2.firstWord ) return true;
    if ( *kt2.firstWord < *kt1.firstWord ) return false;
    // compare by second keyword
    return *kt1.secondWord < *kt2.secondWord;
  }
};

// stores unique keyword tuples and their frequency (count)
using KeywordTupleCounts = map<KeywordTuple, size_t>;

// BinaryPredicate for sorting keyword tuples by frequency
struct CompareKeywordTupleCount
{
  template<typename Iter>
  bool operator()(Iter i, Iter j) // applied on KeywordTupleCounts iterators
  {
    // compare by frequency (count)
    if ( i->second < j->second ) return true;
    if ( j->second < i->second ) return false;
    // compare by keyword tuple (when the count is equal)
    return i->first < j->first;
  }
};

// stores unique keywords from input source
KeywordSet keywords;

// stores unique keyword tuples and their frequency
KeywordTupleCounts tupleCounts;
```

The Algorithm

Loading the keyword database into our data structures (counting search phrase occurrences):

```
ifstream inFile(dbPath, ios_base::in);
auto prevWordIt = end(keywords); // initially invalid
string word;
while ( inFile >> word ) // read each keyword until EOF
{
  // insert/retrieve current word to/from our keyword set
  auto currWordIt = keywords.insert(word).first;

  // increment frequency for this keyword tuple: {prevWordIt, currWordIt}
  if ( prevWordIt != end(keywords) )
    ++tupleCounts[ {prevWordIt, currWordIt} ];

  prevWordIt = currWordIt; // update previous iterator
}
```

Ranking keyword database:

```
// insertion-sort keyword tuples by frequency, using our defined custom predicate
set<KeywordTupleCounts::const_iterator, CompareKeywordTupleCount> topKeywordTuples;
for ( auto it = begin(tupleCounts), endIt = end(tupleCounts); it != endIt; ++it )
  topKeywordTuples.insert(it);

ofstream outFile("top1000.out", ios_base::out);

int cnt = 1000;
for ( auto it = begin(topKeywordTuples), endIt = end(topKeywordTuples);
      it != endIt && cnt > 0; ++it, --cnt )
{
  const auto & tuple = (*it)->first;
  outFile << *tuple.firstWord << " " << *tuple.secondWord
          << " # " << (*it)->second << "\n";
}
```

Query keyword database:

```
cin >> inputKeyword;
// get related searches based on top search phrases containing the input keyword
int matchCount = 10;
for ( auto it = rbegin(topKeywordTuples), endIt = rend(topKeywordTuples);
      it != endIt && matchCount > 0; ++it )
{
  if ( *(*it)->first.firstWord  == inputKeyword ||
       *(*it)->first.secondWord == inputKeyword )
  {
    cout << *(*it)->first.firstWord << " " << *(*it)->first.secondWord
         << " # " << (*it)->second << "\n";
    --matchCount;
  }
}
```

Analyzing the proposed implementation (Solution 1)

PROs

- is a very good showcase for STL usage (serves its didactical purpose)
- is succinct in implementation
- is relatively easy to explain/understand
- uses simple/ordered STL tree data structures: set<>, map<>
- is idiomatic STL usage
- is type-safe and memory safe
- offers good performance characteristics for large data sets
- stores unique keywords (no data duplication)
- offers good memory working set scaling for long search phrases

CONs

- is not cache-friendly (uses tree/cell-based data structures spread all over memory)
- tree data structures (sets/maps) are memory inefficient (a lot of waste in storing 64-bit pointers for tree nodes)
- it uses the notoriously slow C++ I/O streams for data input
- for simplicity, our implementation uses case-sensitive compare for keywords

Solution 2 (faster)

Data structures

For this implementation, the data structures used by the algorithm are **not** designed to store the minimal amount of information in memory, having considerable redundancy in storing the keywords (allows for storing duplicate instances of keywords).

We use an STL `unordered_map<>` container to store all search phrases and their occurrences. We store each keyword pair as a concatenated string "keyword1 keyword2" (map-first) with its corresponding counter (map-second). This is where our data redundancy stems from (duplicated keywords stored in different search pairs).

However, we chose this advanced data structure for our algorithm, because it is a ***hash map***. We leverage this fact for its **speed** in storing a new search phrase (hashing the keyword tuple string) and finding an existing tuple to increment its frequency (in constant time).

Our data structures leverage STL container iterators that are stable (valid) under the algorithm operations used.

Usage of the `CompareKeywordTupleCount` custom binary predicate is optional, because it is not mandatory to perform a *stable* sort (lexicographic) with regards to search phrases (keyword pairs) that have the same rank/frequency.

```
// a keyword tuple is stored as a concatenated string: "keyword1 keyword2"
using KeywordTupleCounts = unordered_map<string, size_t>;

KeywordTupleCounts tupleCounts;

// sorted keyword tuples by frequency (using our defined custom predicate)
vector<KeywordTupleCounts::const_iterator> topKeywordTuples;

// BinaryPredicate for sorting keyword tuples by frequency
// (lexicographical compare - by parts)
struct CompareKeywordTupleCount
{
  template<typename Iter>
  bool operator()(Iter i, Iter j) // applied on KeywordTupleCounts iterators
  {
    // compare by frequency (count)
    if ( i->second > j->second ) return true;
    if ( j->second > i->second ) return false;
    // compare by keyword tuple string (when the count is equal)
    return i->first > j->first;
  }
};
```

The Algorithm

Loading the keyword database into our data structures (counting search phrase occurrences):

```
ifstream inFile(dbPath, ios_base::in);

// A keyword tuple is stored as a concatenated string: "keyword1 keyword2".
// This hash map does all the heavy lifting for us, with very fast storage & retrieval
// of search phrases: O(1) avg. insert/get via the same operator[]
using KeywordTupleCounts = unordered_map<string, size_t>;
KeywordTupleCounts tupleCounts;

string prevWord;
string word;
while ( inFile >> word ) // read each keyword until EOF
{
  // increment frequency for this keyword tuple: {prevWordIt, currWordIt}
  if ( !prevWord.empty() )
    ++tupleCounts[ prevWord + TUPLE_SEP + word ];
  prevWord = word;
}
```

Ranking keyword database:

```
// put all tuple refs (iterators) in a vector, to be sorted
vector<KeywordTupleCounts::const_iterator> topKeywordTuples;
topKeywordTuples.reserve( tupleCounts.size() );
for (auto it = begin(tupleCounts), endIt = end(tupleCounts); it != endIt; ++it)
  topKeywordTuples.push_back(it);

// sort keyword tuples by frequency, using our binary predicate (lexicographic compare)
std::sort( begin(topKeywordTuples), end(topKeywordTuples), CompareKeywordTupleCount());

// OR (faster)

// sort keyword tuples by frequency (not stable with regards to equivalent rank pairs)
std::sort( begin(topKeywordTuples), end(topKeywordTuples),
           [](const auto & i, const auto & j) { return i->second > j->second; } );

ofstream outFile("top1000.out", ios_base::out);

int cnt = 1000;
for ( auto it = begin(topKeywordTuples), endIt = end(topKeywordTuples);
      it != endIt && cnt > 0; ++it, --cnt )
  outFile << (*it)->first << " # " << (*it)->second << "\n";
```

Query keyword database:

```
cin >> inputKeyword;

// get related searches based on top search phrases containing the input keyword
int matchCount = 10;
for ( auto it = begin(topKeywordTuples), endIt = end(topKeywordTuples);
      it != endIt && matchCount > 0; ++it )
{
  const string & tuple = (*it)->first;

  // we need to decompose our concatenated search phrase: "keyword1 keyword2"
  size_t sep = tuple.find(TUPLE_SEP);
  if ( tuple.compare(0, sep, inputKeyword) == 0 ||
       tuple.compare(sep + 1, tuple.size(), inputKeyword) == 0 )
  {
    cout << "   -> " << tuple << " # " << (*it)->second << "\n";
    --matchCount;
  }
}
```

Analyzing the proposed implementation (Solution 2)

PROs

- is succinct in implementation
- is relatively easy to explain (to someone who is familiar with hashed containers)
- is idiomatic STL usage
- is type-safe and memory safe
- offers good performance characteristics for large data sets
- it's very fast (due to ***hash***-based lookup)
- although it duplicates data, its memory usage is lower (on our working database) than [Solution 1], because we have short keywords in our database and [Solution 1] has a lot of memory waste due to tree node 64-bit pointers

CONs

- it stores duplicated keywords
- offers poor memory working set scaling for long search phrases (due to data duplication)
- for simplicity, our implementation uses case-sensitive compare for keywords
- it uses the notoriously slow C++ I/O streams for data input

Alternative solutions and further improvements

- We could use a ***memory mapped file*** to map the keyword database directly into process memory, so that we could avoid using I/O streams and string parsing/processing.

- We could perform a ***partial_sort*** of the keyword tuples (just Top N search phrases) and perform our lookup for suggestions in that pool.

- We could use a much more ***cache-friendly data structure***, like an std::vector<> to store the tuple counts more compactly (**array**). Then, we would sort the array, count adjacent equal pairs; store counts and tuples in another array that we (partially) sort, and read out the range desired.

- Because we are dealing strictly with *English* words, we could cut off (truncate) keywords at **8 bytes** each and store them in a **uint64_t** integer. This is not functionally equivalent, but good enough because most keywords in the database are smaller than ***8 characters*** (ASCII). Using integers instead of strings would be a huge performance boost when performing comparisons and would also be much more space efficient.