Autocomplete Me

[Problem Definition 2](#_Toc467541184)

[Algorithm Work Flow 3](#_Toc467541185)

[Input 3](#_Toc467541186)

[Flowchart 3](#_Toc467541187)

[Project Requirements 5](#_Toc467541188)

[Required Implementation 5](#_Toc467541189)

[Input 5](#_Toc467541190)

[Output 5](#_Toc467541191)

[Test Cases 5](#_Toc467541192)

[Deliverables 6](#_Toc467541193)

[Implementation (60%) 6](#_Toc467541194)

[Document (40%) 6](#_Toc467541195)

[Allowed Codes 6](#_Toc467541196)

[Milestones 6](#_Toc467541197)

[BONUSES 7](#_Toc467541198)

# Problem Definition

Autocomplete is pervasive in modern applications. As the user types, the program predicts the complete *query* (typically a word or phrase) that the user intends to type. Autocomplete is most effective when there are a limited number of likely queries. For example, search engines like: [Google](http://www.google.com) use it to display suggestions as the user enters web search queries; cell phones use it to speed up text input.

Autocomplete speeds up human-computer interactions when it correctly predicts the word a user intends to enter after only a few characters have been typed into a text input field. It works best in domains with a limited number of possible words (such as in command line interpreters), when some words are much more common (such as when addressing an e-mail), or writing structured and predictable text (as in source code editors).

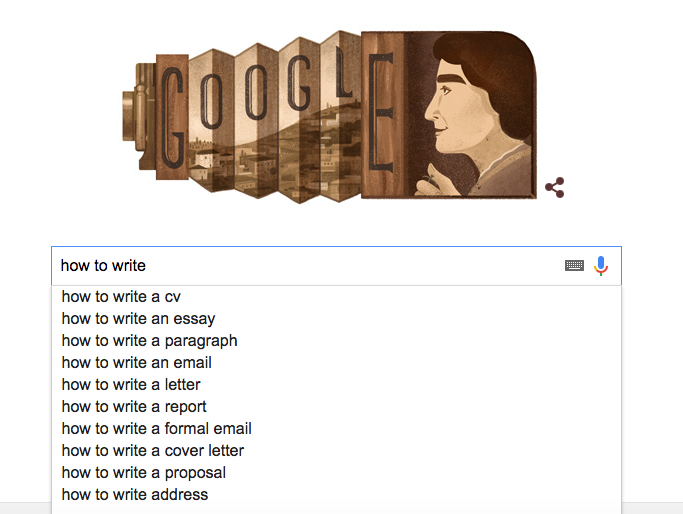


Figure.1 Prefix Search

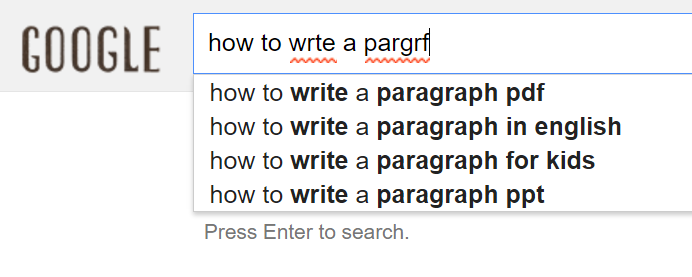
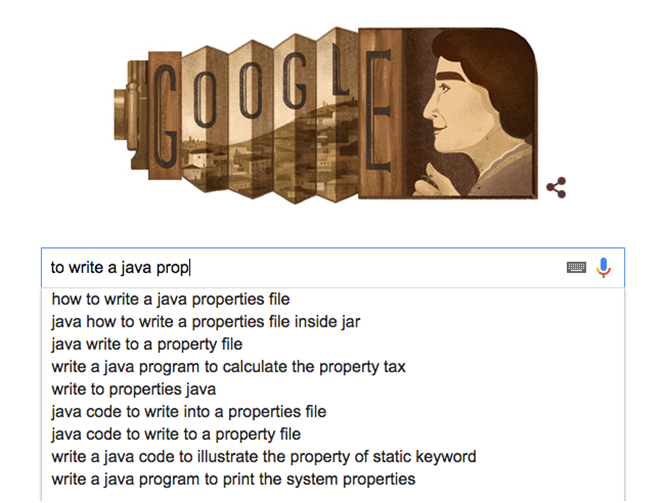


Figure.2 Efficient suggestion



**Required**

(Exact substring)

**BONUS**

(Near matches)

Figure.3 Substring Search

Write a program to implement *autocomplete* from a given set of *N* *terms*, where each term consists of a query string and an associated nonnegative weight. That is, given a word/ sentence, find all queries that contain the given input or find them after auto-correction to the misspelled input, in descending order of weight.

The performance of autocomplete functionality is critical in many systems. For example, consider a search engine which runs an autocomplete application on a server farm. According to one study, the application has only about 50*ms* to return a list of suggestions for it to be useful to the user. Moreover, in principle, it must perform this computation *for every keystroke typed into the search bar* and *for every user*!

Algorithm Work Flow

## Input

1. Search Sentences Dataset (1):

* Consists of *N* pairs of query strings, each is associated with nonnegative weight.
* These weights are determined by historical data, such as box office revenue for movies, frequencies of search queries from other Google users, or the typing history of a cell phone user. For the purposes of this project, you will have access to a set of all possible queries and associated weights
* These queries and weights will **not change**.

1. [English dictionary](Dictionary/dictionary.txt) of words:

* First line contains the number of words in the dictionary (N)
* Next N lines contain the dictionary words, one word per line

## Flowchart

* The below flow chart shows the scenarios that algorithm should follow.
* It is supposed finally to show the **weighted based sorted list** regarding the top matches to the user input.

**Find** all search results that start with the written string “S”, as shown in “figure.1”

Is result found?

No

**Sort** results descending based on weights

(Select 2 different sorting algorithms and compare their performance)

Yes

Spell checking **each** **word** in the written sentence: by looking for it in the given dictionary.

Is each word correct?

Yes

**Search** by the written input “S” as a **substring** of all search sentences, as shown in “figure.3”

No

Show the suggested word(s) for the user on real time (as in “figure.2”)

Is result found?

No

Display results to the user

The user starts to write the required sentence (S)

(single OR multiple word string)

Yes

Display “No Suggestions Found!”

# Project Requirements

## Required Implementation

|  |  |
| --- | --- |
| **Requirement** | **Performance** |
| 1. Create GUI for the user that matches the whole scenario. | |
| 1. Efficient prefix search on the given dataset to find the user input. | **Time:** should be **less than** **O(N)**, N is number of sentences (query strings) |
| 1. Sort the filtered results by their corresponding weights (Switchable from the GUI). | **Time:** Compare between two different “Complexity” sorting algorithms |
| 1. Efficient algorithm to check whether a word is correctly spelled or not. | **Time:** should be **less than** **O(M)**, M is number of words in the dictionary |
| 1. Efficient suggestion algorithm to correct the words spelling based on the dictionary. |  |
| 1. Substring searching algorithm. |  |
| 1. Display the list of suggestions and results for the user. | |

## Input

1. **File1:** [English dictionary](Dictionary/dictionary.txt) of words.
2. **File2:** Search Sentences Dataset

* The file consists of an integer *N* followed by *N* pairs of query strings and nonnegative weights.
* One pair per line, with the weight and string separated by a tab.
* A weight can be any integer between 0 and 2^63 − 1. A query string can be an arbitrary sequence of Unicode characters, including spaces (but not newlines).

1. User input query through the GUI.

## Output

1. GUI based application to enable the user to:
2. Search
3. Choose suggested word(s)
4. Show top matched results
5. Select between two different sorting algorithms
6. Top matched results sorted by their weights.
7. Suggestions list of the most nearest words, when the word isn’t included in the dictionary.

## Test Cases

* [**Sample Tests:**](Sample%20Test)
* Few test cases with small values that can be traced.
* THREE testing scenarios
* Use the sample dictionary file “.\Sample Dictionary\”
* Use the given search links file of each scenario “.\Search Links\”
* Write the queries of each scenario and check the expected outputs “.\Search Queries\”
* [**Complete Tests:**](Complete%20Test)
* Large cases for massive testing.

# Deliverables

## Implementation (60%)

1. Create GUI for the user.
2. Efficient prefix search.
3. Compare between two different “Complexity” sorting algorithms for weights- sorting.
4. Efficient algorithm to check whether a word is correctly spelled or not.
5. Efficient suggestion algorithm.
6. Substring search algorithm.
7. Display the list of suggestions and results for the user.

## Document (40%)

1. Algorithms description and code of each stage of the flow chart
   1. Prefix Search
   2. Sort
   3. Suggestion
   4. Substring Search.
2. Detailed analysis of the above codes.

## Allowed Codes

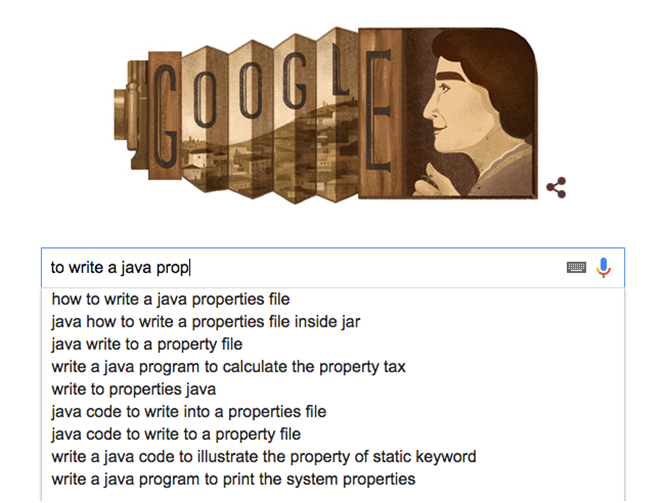
* No external code is allowed.

# Milestones

|  |  |  |
| --- | --- | --- |
|  | **Deliverables** | **Due to** |
| **Milestone1** | 1. Create GUI for the user. 2. Efficient prefix Search. 3. Use and compare between two different “Complexity” sorting algorithms for weights. 4. Display the list of results. 5. Efficient algorithm to check if a word is correctly spelled. 6. Documentation I | Week12: start at 10-12-2016  [During LABS] |
| **Milestone2** | 1. Efficient suggestion algorithm. 2. Substring searching algorithm. 3. Display the list of suggestions. 4. Documentation II | Final Delivery  Lab Exam week start at SAT 31-12-2016 |
| **For Milestone1:**   * + **MUST** deliver the required tasks and **ENSURE** it’s worked correctly   + **MUST** deliver the **part of the documentation** that is related to the Milestone (printed document)   + **MUST** deliver in the **section time of the majority members** (e.g. if a group consists of 3 members from sec.1 and two members from sec.2, they should deliver in the time of sec.1) | | |

# BONUSES

1. Efficient substring searching.
2. Don’t search again when user completes typing.
3. Beside the substring search, search for the nearest set of matches that contain the set of words with any order, as shown in the figure below



**Required**

(Exact substring)

**BONUS**

(Near matches)

1. Any other efficient recommendations are welcomed.