Prerna Satija
prernasatija24@gmail.com

Code:
Programming language used: Java
The project folder consists of 5 Java files:

MainServer.java
It is the main program that reads the file to construct the Index structure. It provides the index structure to create the QueryServer. And then handles user queries and returns top 10 results.

Pair.java
It is a model to store the input Java variable and its score. It implements Serializable and Comparable interfaces.

IndexConstruction.java
It is the *Construction Program* to solve problem 1. It produces the index structure which stores each input Pair and a corresponding Id to it. It provides methods to serialize and deserialize the index structure. It also provides a method to sort the list of Pairs according to score.

QueryServer.java
It is the *Query Server Program* to solve problem 2. It takes in the IndexConstruction "*index"* and provides the "*find*" api for query search. The implementation is based on Compressed Trie data structure. It has a inner class, TrieNode, which is the node for the Compressed Trie.

TestServer.java
JUnit class used to test the query search functionality. It has 5 testcases.

Analysis & Research:
As the given problem is similar to a prefix-matching problem, I thought of constructing a Trie data structure to store all the input data and use it to handle the queries. To make it more efficient I compressed the Trie to form a CompressedTrie. A Compressed Trie is a Trie where a parent and child can be combined to a single node if that is the only child. In order to better understand CompressedTrie, I used the following sources:

1. http://en.wikipedia.org/wiki/Trie#Compressing_tries
2. http://fbim.fh-regensburg.de/~saj39122/sal/skript/progr/pr45102/Tries.pdf
3. Willard, Dan E. "New trie data structures which support very fast search operations." *Journal of Computer and System Sciences* 28.3 (1984): 379-394.
4. http://c2.com/cgi/wiki?StringTrie

I modified the Compressed Trie data structure algorithm for this specific problem, as we need to handle the '_' search. So, for each input java variable string, we add that string and all substrings starting with '_'. Example: For string *s = "i_am_an_example"*, I add *s* and also "*_am_an_example*", "*_an_example*", and "*_example*" to the Trie. Java variables are case sensitive so my implementation also treats the upper and lower case variables differently.

Prerna Satija
prernasatija24@gmail.com

Running Time of Query Search:
For a given query of length '*m*', the program will search the node in *O(m)* running time. Extracting all the "*names*" under this node will take *O(n)* running time, where '*n*' is the number of "*names*" under this node. Sorting these output "*names*" using Collections sort will take *O(nlog(n))* running time. In practical scenarios, at each level we are reducing the number of output names logarithmically.

Hence, *n = log(N), where N is the number of input names.*
Therefore, the running time is *O(logN.log(logN)))* which is sub-linear

To further reduce the running time and make it more efficient we can get the top results using a Heap data structure. As the problem specifies that only top 10 results are required then this can be achieved using a fixed length Heap. By doing this, the running time will reduce to *O(log(logN))*. Note: Heap is not implemented in the program that I have submitted.

Improvement:
In the updated code, TrieNode stores the list of top 10 results corresponding to the node value, which is done when the Compressed Trie is constructed. Hence, when a query is entered, the top ten results are directly fetched once the proper TrieNode for the query is found. So, the improved running time for query search will depend on the query length and will be *O(m)*, where '*m*' is the length of the query.

How to run the program:
Download the folder and put in your workspace.
The *MainServer* reads data from "*sample.txt*". The data in this file is expected to be in format "*variable_name,score*".

It took me approximately 9 hours to complete it. The breakdown is follows:
Analysis & Research: 1 hour
Implementation: 5 hours
Testing & bug fixing: 2 hours
Preparing the doc: 30 min

I've written all the code myself and no third party code is used.