ALGORITHM FOR COMPUTING USER KNOWLEDGE DEGREE:

algorithm USER_KNOWLEDGE_DEGREE ():

```
keywords = AUTOMATIC_KEYWORD_EXTRACTION(answers)
no_of_replies = COUNT(GET_USER_REPLIES(user))
no_of_interactions = NORMALIZE(no_of_replies)
```

answer1 = """When data is stored on disk based storage devices it is stored as blocks of data. These blocks are accessed in their entirety making them the atomic disk access operation. Disk blocks are structured in much the same way as linked lists both contain a section for data a pointer to the location of the next node (or block) and both need not be stored contiguously. Due to the fact that a number of records can only be sorted on one field we can state that searching on a field that isnt sorted requires a Linear Search which requires N/2 block accesses (on average) where N is the number of blocks that the table spans. If that field is a non-key field (i.e. doesnt contain unique entries) then the entire table space must be searched at N block accesses. Whereas with a sorted field a Binary Search may be used this has log2 N block accesses. Also since the data is sorted given a non-key field the rest of the table doesnt need to be searched for duplicate values once a higher value is found. Thus the performance increase is substantial.Indexing is a way of sorting a number of records on multiple fields. Creating an index on a field in a table creates another data structure which holds the field value, and pointer to the record it relates to. This index structure is then sorted allowing Binary Searches to be performed on it. The downside to indexing is that these indexes require additional space on the disk since the indexes are stored together in a table using the MyISAM engine this file can quickly reach the size limits of the underlying file system if many fields within the same table are indexed. When should it be used? Given that creating an index requires additional disk space (277778 blocks extra from the above example) and that too many indexes can cause issues arising from the file systems size limits careful thought must be used to select the correct fields to index. Since indexes are only used to speed up the searching for a matching field within the records it stands to reason that indexing fields used only for output would be simply a waste of disk space and processing time when doing an insert or delete operation and thus should be avoided. Also given the nature of a binary search the cardinality or uniqueness of the data is important. Indexing on a field with a cardinality of 2 would split the data in half whereas a cardinality of 1000 would return approximately 1000 records. With such a low cardinality the effectiveness is reduced to a linear sort and the query optimizer will avoid using the index if the cardinality is less than 30% of the record number effectively making the index a waste of space."""

answer2 = """What would happen without an index? Database software would literally have to look at every single row in the Employee table to see if the Employee_Name for that row is Abc. And because we want every row with the name Abc inside it we can not just stop looking once we find just one row with the name Abc because there could be other rows with the name Abc. So every row up until the last row must be searched - which means thousands of rows in this scenario will have to be examined by the database to find the rows with the name Abc. This is what is called a full table scan How a database index can help performance The whole point of having an index is to speed up search queries by essentially cutting down the number of records or rows in a table that need to be examined. An index is a data structure (most commonly a B- tree) that stores the values for a specific column in a table. How does B-trees index work? The reason B-trees are the most popular data structure for indexes is due to the fact that they are time efficient - because look-ups deletions and insertions can all be done in logarithmic time. And another major reason B- trees are more commonly used is because the data that is stored inside the B- tree can be sorted. The RDBMS typically determines which data structure is actually used for an index. But in some scenarios with certain RDBMSs you can actually specify which data structure you want your database to use when you create the index itself. How does a hash table index work? The reason hash indexes are used is because hash tables are extremely efficient when it comes to just looking up values. So queries that compare for equality to a string can retrieve values very fast if they use a hash index. For instance the query we discussed earlier could benefit from a hash index created on the Employee_Name column. The way a hash index would work is that the column value will be the key into the hash table and the actual value mapped to that key would just be a pointer to the row data in the table. Since a hash table is basically an associative array a typical entry 0x28939 is a reference to the table row where Abc is stored in memory. Looking up a value like Abc in a hash table index and getting back a reference to the row in memory is obviously a lot faster than scanning the table to find all the rows with a value of Abc in the Employee_Name column. The disadvantages of a hash index Hash tables are not sorted data structures and there are many types of queries which hash indexes can not even help with. For instance suppose you want to find out all of the employees who are less than 40 years old. How could you do that with a hash table index? Well its not possible because a hash table is only good for looking up key value pairs - which means queries that check for equality. What exactly is inside a database index? So now you know that a database index is created on a column in a table and that the index stores the values in that specific column. But it is important to understand that a database index does not store the values in the other columns of the same table. For example if we create an index on the Employee_Name column this means that the Employee_Age and Employee_Address column values are not also stored in the index. If we did just store all the other columns in the index then it would be just like creating another copy of the entire table - which would take up way too much space and would be very inefficient. How does a database know when to use an index? When a query is run the database will check to see if there is an index on the column(s) being queried. Assuming the Employee_Name column does have an index created on it the database will have to decide whether it actually makes sense to use the index to find the values being searched - because there are some scenarios where it is actually less efficient to use the database index and more efficient just to scan the entire table. What is the cost of having a database index? It takes up space - and the larger your table the larger your index. Another performance hit with indexes is the fact that whenever you add delete or update rows in the corresponding table the same operations will have to be done to your index. Remember that an index needs to contain the same up to the minute data as whatever is in the table column(s) that the

```
index covers. As a general rule an index should only be created on a table if the data in
the indexed column will be queried frequently."""

answer3 = """Since then I gained some insight about the downside of creating indexes:
if you write into a table (UPDATE or INSERT) with one index you have actually two
writing operations in the file system. One for the table data and another one for the
index data (and the resorting of it (and - if clustered - the resorting of the table data)).
If table and index are located on the same hard disk this costs more time. Thus a table
without an index (a heap) would allow for quicker write operations. (if you had two
indexes you would end up with three write operations and so on) However defining
two different locations on two different hard disks for index data and table data can
decrease/eliminate the problem of increased cost of time. This requires definition of
additional file groups with according files on the desired hard disks and definition of
table/index location as desired. Another problem with indexes is their fragmentation
over time as data is inserted. REORGANIZE helps you must write routines to have it
done. In certain scenarios a heap is more helpful than a table with indexes e.g:- If you
have lots of rivalling writes but only one nightly read outside business hours for
reporting. Also a differentiation between clustered and non-clustered indexes is rather
important."""

document1 = tb(answer1.lower())

document2 = tb(answer2.lower())

document3 = tb(answer3.lower())

IntTema = 0

M = answer1.count("index")
print("Count of word index is: %d" % M)

bloblist = [document1, document2, document3]

for i, blob in enumerate(bloblist):
   print("Top words in document {}".format(i + 1))
   scores = {word: tfidf(word, blob, bloblist) for word in blob.words}
   sorted_words = sorted(scores.items(), key=lambda x: x[1], reverse=True)
   for word, score in sorted_words[:1000]:
      if(word in keywords):
        print("Word: {}, TF-IDF: {}".format(word, round(score, 10)))
        IntTema = IntTema + score * M

print("IntTema value is: %f" % IntTema)
C = IntTema * no_of_interactions

print("Knowledge of user is: ")
print(C)
```

```
algorithm NORMALIZE(no_of_replies):
normalized_value = (float)(math.sqrt(math.sqrt(math.log(no_of_replies+1))))
return normalized_value

algorithm TERM_FREQUENCY(word, blob):
return (float)(blob.words.count(word)) / (float)(len(blob.words))

algorithm N_CONTAINING(word, bloblist):
    return (float)(sum(1 for blob in bloblist if word in blob))

algorithm INVERSE_DOCUMENT_FREQUENCY(word, bloblist):
    return (float)(math.log(len(bloblist)) / (float)(1 + n_containing(word, bloblist)))

algorithm TERM_FREQUENCY_INVERSE_DOCUMENT_FREQUENCY(word, blob, bloblist):
    return (float)((float)(tf(word, blob)) * (float)(idf(word, bloblist)))
```

FLOW OF ALGORITHM WITH SAMPLE INPUT:

```
Count of word index is: 13

Top words in document 1

Word: disk, TF-IDF: 0.0046354949

Word: fields, TF-IDF: 0.0046354949

Word: index, TF-IDF: 0.0034766212

Word: records, TF-IDF: 0.0030903299

Top words in document 2

Word: index, TF-IDF: 0.0104010631

Word: records, TF-IDF: 0.0004333776

Top words in document 3

Word: index, TF-IDF: 0.0063284118

Word: disk, TF-IDF: 0.0016875765

IntTema value is: 0.450949

Knowledge of user is: 0.489317905107
```