1) Pseudo Code for add()

add (HashTable T, tuple T)

O(1) - Calculate hashValue of the key of the tuple

O(1) - Go to the array list pointed by the T[hashValue]

O(nK) - for each element in the array list

O(K) • If the array list contains a tuple with the same key & value, only increment the count of the tuple in the array by 1.

• If we reach the end of the ArrayList and the tuple was not found, then we add the tuple to the array list

O(n) - calculate loadFactor of the hashTable T

O(1) - If (loadFactor > 0.7)

• create a new hashTable T1 of size that is the prime which is immediately greater than twice the size of hashTable T.

• create a new HashFunction

O(nK) - for each element in the HashTable T

• find the hash value of the key & add the tuple to the array list at that location.

∴ add(hashTable T, tuple t) ∈ O(nK)

- However, on the <u>average case</u> we consider the add operation to take O(K) time.

Pseudo code for search (hashTable T, tuple t)

search (hashTable T, tuple t)

O(1) - Calculate hash value of the key of the tuple.

- go to the array list pointed by the T[hashValue]

O(K) - for each tuples in the Array List

O(K) - if s.equals(t)

return t.getCount()

- else return 0

$\therefore$ search(T, t) $\in$ O(K)

4)

	Similarity	Runtime
Brute force	0.39867774	62.748 s
Hash String	0.39867774	24.448 s
Hash Code	0.39868575	4.842 s

5) Hash Code Similarity produces the shortest runtime.
Brute Force Similarity produces the longest runtime.

- Although, its not true in my implementation, but the runtime for Hash String & Hash Code should be near to each other.
- Brute Force is obviously going to take the maximum time. The number of distinct elements in S1 is 47535 (with shingle length 8) & in S2 is 50271. B
- Brute force would have to compare each of the 8-length shingles in S1 to every other shingle in S2, thus producing the highest run time.
- Hash String compares the strings individually as well, but only the strings which have been hashed in the same bucket. This is a mere fraction of the total number of strings.
- In this particular example, the maxload() returned a value of 6. That means even in the very worst case, the Hash String was comparing each 8-length string with only 6 strings. Contrast this with the brute force where each string was being compared with 50,000+ strings & it becomes evident that Hash String would run much faster.
- Hash Code on the other hand does not compare strings at all & deals only with integers. This is a constant time operation & therefore takes absolutely minimum amount of time.

6) The Brute Force & HashString return the same value for similarity. However, the HashCode class returns a different value for similarity

- This because in HashCode class we are comparing the hash values obtained by rollover hash function and not the strings themselves.
- Therefore, we might run into a case where the rolling hash value obtained from 2 different strings ends up being the same. In such a case, although the strings are different, we would still end up calculating them as similar cases & thus lead to a higher value.

3) Brute Force Similarity:

a) Data Structures:

- String shak1 & shak2 are split into equal length strings of length "slength" each. Each of these strings of length slength are stored in an arrayList of Strings.
- In my code, this corresponds to arrayL S1_array and arrayL S2_array. Both of these contain repetitions.
- The unique elements from both of the arrays are merged to form a single ArrayList arrayU in my code. This contains distinct elements and is the same as set U given in the problem.
- We also define a new Class CountingTuples that stores the String Value & the number of times that particular string appears in ArrayList S1_array & S2_array.
- We define an ArrayList of class CountingTuples called myCount
- The basic idea behind this is that when we want to find out the count of a particular string, instead of querying the entire S1_array / S2_array, we instead query the ArrayList myCount which has already stored the string & count for that string in each of the arrays S1_array & S2_array for us.

b) Pseudo Code:

O(n-K)+O(m-K) 1> Split string S_1, S_2 into length of s length each and store them in an array list S1_array & S2_array respectively.

2> Calculate length Of S1 :

O(1) — 2.1> Initialise 2 array lists markedS1 & markedS2

O(n) — 2.2> for each element in S1_array {

O(nK) — if(element is not present in markedS1){ add element to markedS1

O(n) — for(each element in S1_array starting from the current){ - count the number of times the element occurs in S1_array.}

O(Kn(m+n)) { O((m+n)K) — check if there is a tuple of Value string in myCount array List

O(1) — - Create a new tuple of Class CountingTuples with fields Value(string itself), count_In_S1 = 0, count_In_S2 = 0 if no element found in prev. step

O(1) — - set count_In_S1 to the count obtained in the previous step.

O(1) - sum += count * count // Calculate a running sum to give vector length

}

vectorSum1 = Math.sqrt(sum)

O(Km(m+n)) 3> Similarly calculate length Of S2()

4> Similarity :

O(n+m) 4.1> for(each Tuple i in myCount arrayList

sum += count_In_S1 * count_In_S2.

O(1) Similarity = sum / (length of S1 * length of S_2)

∴ Run Time ∈ O(mnK)

Hash Code Similarity :

a) Data Structure :

- String shak1 and shak2 are split into equal length strings of length slength each. Each of these strings of length slength are stored in an arrayList of strings.
- This corresponds to the array S1_array & S2_array. Both of these contain repetitions.
- The unique elements from both of these arrays are merged to form a single array list arrayU. This set contains distinct elements and is equivalent to set U given in the problem.
- We create 2 hash tables hashTable_S1 and hashTable_S2 that contain unique elements from the multisets S1_array & S2_array respectively.
- These hash tables are built using the HashTable Class and therefore store elements as Tuples (with value of string and count)
- Basic Idea is that for every string value stored in the Array U, we make 2 searches. The first one is in the hashTable-S1 that returns the count of the string in S1_array. The second one is in hashTable-S2 that returns the count of that string in the S2_array.
- Once we have these counts, we can calculate length of S1(), length of S2 & similarity()

b) Pseudo Code:

O(m-k) 1) Split string S_1, S_2 into lengths of slength each and store
O(n-k) them in an arraylist S1_array & S2_array respectively.

2) Create hashTable_S1 from the strings stored in S1_array

2.1) Initialize an instance of the hashTable class with hash table size of 8 => HSS_Table_1

O(k) 2.2) Compute & store an exponent array that contains powers of the alpha value upto $(alpha)^{slength}$ => exponent[]

O(m) 2.3) Create a new array which stores each character of the string
shak1 => 8 array1[]

2.4) Calculate the first hashValue by doing the following:

O(k) - for (i=0 ; i < slength ; i++)
{

char c = array1[i].charAt[0] // take the first
// single length string from the array1 and
// convert it into a char
asciiValue = (int) c ; // calculate the ascii value
// of the character.
h1 = h1 + (asciiValue * exponent[i])
↳ initialized to 0.

add() takes constant time since we are comparing primitives

}

O(1) 2.5) Create a new Tuple and add it into the hashTable_S1 with values (h1, S1_array[0])

O(m-k) 2.6) Calculate remaining values by using Rollover Hashing.

for (int i = 1 ; i < array1.length - slength + 1 ; i++) {
h_next = (h1 - Substract) * alpha + asciiValue_add

h_next: next hash rollover value
Substract: ascii value of the character to be removed
asciiValue_add: ascii value of the character to be added.

O(1) - create a new tuple and add it to the hashTable_S1 with values (h_next, S1_array[i])

}

return hashTable_S1.

O(n) 3> Use the same process outlined in step 2 to create the hashTable_S2 from the contents of the array_S2.

4> Create Array U that contains distinct elements from the union of hashTable_S1 and hashTable_S2.

O(m) 4.1> Copy all the tuple.Key() of hashTable_S1 into Array U.

O(n) 4.2> For each tuple in the hashTable_S2 {

- get the Key of the tuple

Average O(1) - search for the Key in hashTable_S1

- If Key does not exist in hashTable_S1, add the Key to the Array U

}.

5> Calculate length Of S1()

O(m+n) 5.1> for each Key in the Array U {

Average: O(1) - search for the tuple with the same Key in hashTable_S1

- return the count of the tuple

- calculate running sum

sum += (count * count)

}

Vector Sum = Math.sqrt(sum);

O(m+n) 6> Calculate length of S2() similarly.

7> Calculate Similarity()

O(m+n) 7.1> for each Key in the Array U {

O(1) - Search for the tuple with the same Key in hashTable_S1

- return the count of the tuple = count_1

O(1) - Search for the tuple with the same Key in hashTable_S2

- return the count of the tuple = count_2.

sum += count_1 * count_2.

}

RunTime ∈ O(m+n)

Hash String Similarity

— DataStructures:

- Strings shak1 and shak2 are split into equal length strings of length slength each. Each of these strings are stored in an arrayList of strings
- This corresponds to the array S1_array & S2_array. Both of these contain repetitions.
- We create 2 hash tables hashTable_S1 and hashTable_S2 that contain unique elements from the multi-set S1_array & S2_array respectively
- These hash tables are built using the HashTable class & therefore store elements as tuples (with value of string & count)
- We create another hash table hashTable_U. The unique elements from both the hashtables hashTable_S1 and hashTable_S2 are merged to form a distinct set of elements. This is the same as the set U in the problem.
- The basic idea is that we are going to iterate through the hashTable_U and foreach element in table_U we are going to find out the count per that element by searching in hashTable_S1 and hashTable_S2.

Pseudo Code for Hash String Similarity

O(m-K) 1) Split the strings s1, s2 into lengths of slength each & store
O(n-K) them in an array list S1_array & S2_array respectively.

2) Create hashTable_S1 from the strings stored in the s1_array

2.1) Initialize an instance of the hashTable class with hashTable size of 8 ⇒ HSS_Table_1.

O(K) 2.2) Compute & store an exponent array that contains powers of the alpha values upto $(alpha)^{slength}$ ⇒ exponent[]

O(m) 2.3) Create a new array which stores each character of the string shak1 ⇒ array1[]

2.4) Calculate the first hash Value by doing the following:

O(K) - for (i=0; i < slength ; i++)
{
char c = array1[i].charAt[0] // take the first
// single length string from the array 1 and convert it
// into a char.
asciiValue = (int) c ; // calculate the ascii value of
// the character.
h1 = h1 + (asciiValue * exponent[i])
↳ initialized to 0.
}

since we will be comparing the value() and the Key() with each tuple

O(K) 2.5) Create a new Tuple with values (h1, s1_array[0]) and add it to the hashTable_S1.

2.6) Calculate remaining values using rollover hashing.

O(m-K) — for (int i=1 ; i < array1.length - slength + 1 ; i++) {
h_next = (h1 - substract) * alpha + asciiValue_add.

- h_next: next hash value
- rollover: prev value of hash rollover
- substract: ascii value of the character to be removed
- asciiValue_add: ascii value of the character to be added.

O(K) - create a new Tuple t (h_next, S1_array[i])
and add it to the hashTable_S1
}
return hashTable_S1.

3> Use the same process outlined in step 2 to create the hashTable_S2 from the contents of the array_S2.

4> Create hash_Table_U

4.1> Initialize an instance of the hashTable class with a hash table of size 8 ⇒ HSS_Table_U

O(m) 4.2> for each tuple in the hashTable_S1

average O(1) - add the tuple to hashTable_U

time.

O(nK) 4.3> for each tuple in the hashTable_S2. {

O(K) - search the tuple.getValue() in the hashTable_U

O(1) - if hashTable_U does not contain the tuple then add the tuple to the hashTable_U.

}

5> Calculate length Of S1 ()

5.1> for each tuple in the hash_Table_U {

O((m+n)K) O(K) - search the tuple.getValue() in the hashTable_S1.

- return the tuple.getCount() ⇒ count

- calculate the running sum

sum += (count * count)

}

5.2> vector Length = Math.sqrt(sum)

O((m+n)K) 6> Calculate length of S2() similarly.

7> Calculate Similarity ()

O((m+n)K) 7.1> for each tuple in hashTable_U {

O(K) - search the tuple.getValue() in the hashTable_S1

- return the count of the tuple = count_1

O(K) - search the tuple.getValue() in the hashTable_S2

- return the count of the tuple = count_2.

sum += count_1 * count_2.

}

Run Time ∈ O((m+n)K)