# Problem 1:

**Indices Of Words In Text String**

Given some *text* and a bunch of *words*, find where each of the *words* appear in the *text*. Text may contain spaces, words may not.

For every given *word* you need to return a list of (zero-based) indices of where that word starts in the *text*. If a word isn’t found in the *text*, return [-1] for that word.

**Example**

Input:

*text* = “you are very very smart”

*words* = [“you”, “are”, “very”, “handsome”]

Output:

[ [0], [4], [8, 13], [-1] ]

“you” is found in the given text starting at the index 0.

“are” is found in the given text starting at the index 4.

“very” is found in the given text two times, starting at the indices 8 and 13.

“handsome” isn’t found in the given text.

Constraints:

* *Text* and *words* may contain a-z, A-Z, 0-9, “$”, “#”, “@”, “?”, “;”.
* *Text* may contain spaces, too, but never two or more spaces consecutively. Spaces separate words in the *text* string.
* *Text* won’t start or end with a space.
* Indexing of characters in *text* is zero-based.
* *Words* list will contain unique strings.
* 1 <= number of characters in *text* <= 1000000
* 1 <= number of *words* <= 100000
* 1 <= length of any word in *words* or in *text* <= 10

# Solution

We provided three solutions.

**1) brute\_force\_solution.java**

We literally compare each word from *words* with every word from the *text*. When the two are equal we take note of the start index of the word in the *text*.

**Time Complexity:**

O(*n\*w\*l*).

Let *n* be the number of words in *text*,

*w* be the number of *words* and

*l* be the maximum length of a word.

Processing each pair of words takes O(*l*) time as we compare them character by character. We compare each of *n* words with every one of *w* words for the total time complexity of O(*n\*w\*l*).

**Auxiliary Space Used:**

O(*n+w*).

The list of lists that we return takes O(*w+n*) space. Since *words* are unique, any given word from *text* can match at most one word from *words* so the total number of indexes in the returned list of lists won’t exceed *n* and we know that the outer list has exactly *w* lists, giving us a total of O(*w + n*).

**Space Complexity:**

O((*n+w)\*l*).

Adding up O(*w\*l*) of *words*, O(*n\*l*) of *text* and O(*w+n*) of the auxiliary space we get O((*n*+*w)\*l*).

**2) optimal\_solution1.java**

In this solution we preprocess *text* and create its index, see “HashMap textIndex” variable. By the time that’d done, each word from the *text* has the list of its starting indices pre-compiled. All that’s left is to look up those lists of indexes for every word from *words*.

**Time Complexity:**

O((*n*+*w*)\**l*).

It takes O(*I*) time to calculate hashcode or to compare two strings up to *l* characters long. Thus populating the hashmap with *n* words will take O(n\**l*), making *w* searches in that hashmap will take O(*w*\**l*). Total time is the sum of those: O(*n*\**l*) + O(*w*\**l*) = O((*n*+*w*)\**l*)

**Auxiliary Space Used:**

O(*(n\*l)+w*).

Hashmap which we pre-compute takes O(*n\*l*) space.

The list of lists that we return takes O(*w+n*) space. Summing up the two gives O(*n\*l*) + O(*w+n*) = O(*(n\*l)+w*).

**Space Complexity:**

O(*(n+w)\*l*).

*Text* input takes O(*n\*l*) and *words* take O(*w\*l*). Adding up those two and the auxiliary space we get the total space complexity: O(*n\*l*) + O(*w\*l*) + O(*(n\*l)+w*) = O(*(n+w)\*l*).

**3) optimal\_solution2.java**

In this solution we use a trie (prefix tree), see <https://en.wikipedia.org/wiki/Trie>. First we insert all words from the *text* into the trie. Then we look up every word from *words* in the trie.

Although this solution has the same worst case time and space complexity as the hashmap based optimal\_solution1.java, it will utilize less space when many words share common prefixes.

In the actual interview many interviewers will prefer to hear the trie based solution to the hashmap based one.

For more hints on hashmap vs. trie based algorithms, see

<https://stackoverflow.com/questions/245878/how-do-i-choose-between-a-hash-table-and-a-trie-prefix-tree>

**Time Complexity:**

O(*(n+w)\*l*).

Insert and search operations in the trie take O(*l*) time each.

The algorithm makes *n* insertions and *w* searches.

**Auxiliary Space Used:**

O(*n\*l* + *w*).

Trie takes O(*n\*l*) space and the list of lists that we return takes O(*w+n*). Adding that up we get O(*n\*l* + *w*).

**Space Complexity:**

O(*(n+w)\*l*).

*Text* input takes O(*n\*l*) and *words* take O(*w\*l*). Adding auxiliary space to that we get the total space complexity: O(*n\*l*) + O(*w\*l*) + O(*n\*l* + *w*) = O(*(n+w)\*l*).

# Problem 2:

**Minimum Window Substring**

You are given alphanumeric strings *s* and *t*. Find the minimum window (substring) in s which contains all the characters of t.

**Example One**

Input:

AYZABOBECODXBANC

ABC

Output: BANC

The minimum window is "BANC", which contains all letters - A B and C. We cannot find a window of smaller length than “BANC”.

**Example Two**

Input:

BACRDESDFBAER

BAR

Output: BACR

Here, we can see that there are 2 smallest windows - “BACR” and “BAER”. However, the output is “BACR” because it is the leftmost one.

Output: Return a string *result*, which is the minimum window (substring) in string *s* that contains all characters of string *t*. If no such window exists, then return an “-1” string and if there are multiple minimum windows of the same length, then return the leftmost window.

Constraints:

* 1 <= length*(s)* <= 100000
* 1 <= length(*t*) <=100000

**Custom Input**

Input Format: The first line of input contains string *s*. The next line contains string *t*. If *s = “azisdflc”* and t *= “zsd”* then input should be:

*azisdflc*

*zsd*

Output Format: Output in a single line a string which is the minimum window that contains all the characters of string *t*.

For input *s = “azisdflc”* and t *= “zsd”*, output will be: zisd

# Solution:

# We provided two solutions.

**1) brute force solution.java**

We check whether each substring of string *s* is a valid window or not. If we find it to be a valid window, we update our result accordingly.

**Time Complexity:**

O(*n*^3) where *n* is the length of string s.

As we are checking for all substrings and as there are O(*n*^2) substrings and we take O(*n*) time to check whether the particular substring can be a valid window, so total complexity is O(*n*^3).

**Auxiliary Space Used:**

O(*1*).

We create frequency arrays of size O(*128*) to count the occurrence of each character present in strings *t* and *s*. So overall complexity is O(*1*).

**Space Complexity:**

O(*n+m*) where *n* is the length of string *s* and *m* is the length of string *t*.

For storing input it will take O(*n+m*), as we are storing two strings of length *n* and length *m* and the auxiliary space used is O(*1*) hence total complexity will be O(*n+m*).

Note: We could use an array of length 62 (with some mapping) instead of 128, but this is a general solution which works for the input string containing any ASCII characters.

**2) optimal\_solution.java**

In this approach, we create an array named frequency to keep a count of occurrences of each character in string *t* (O(*length of t*)). Now we start traversing the string S and keep a variable “cnt” which increases whenever we encounter a character present in string *t*. When the value of count reaches the length of *t*, this substring contains all the characters present in string *t*. We try removing extra characters as well as unwanted characters from the beginning of the obtained string. The resultant string is checked whether it can become the minimum window, and the answer is updated accordingly.

This algorithm uses the 2 pointer method, which is widely used in solving various problems. You can refer <https://www.geeksforgeeks.org/two-pointers-technique/> article to get an idea about this method as well as see related problems where this method can be applied.

**Time Complexity:**

O(n) where *n* is the length of string *s*.

Since each character of string *s* is traversed at most 2 times, the time complexity of the algorithm is O(*n*) + O(*m*).

**Auxiliary Space Used:**

O(*1*).

We are creating 2 frequency arrays of size *128*, which use extra space O(*128*) + O(*128*). Hence it is O(*1*).

**Space Complexity:**

O(*n+m*) where *n* is the length of string *s* and *m* is the length of string *t*.

For storing input it will take O(*n+m*), as we are storing two strings of length *n* and length *m* and the auxiliary space used is O(1) hence total complexity will be O(*n+m*).

Note: We could use an array of length 62 (with some mapping) instead of 128, but this is a general solution which works for the input string containing any ASCII characters.

# Problem 3:

*\* Print A String Sinusoidally  
 \*  
 \* This is a string puzzle problem disguised as a programming problem.  
 \* Also called "SnakeString". For example, the phrase "Google Worked"  
 \* should be printed as follows (where ~ is the word separator):  
 \*  
 \* o ~ k  
 \* o g e W r e  
 \*G l o d  
 \*  
 \* The length of each row should be the same, i.e. there should be two  
 \* spaces at the end of the first line, one space at the end of the second line,  
 \* and zero spaces at the end of the third line.  
  
 \* For the function: s = Google Worked  
 \* First character “G” is printed on the third row, first column.  
 \* Second character “o” is printed on the second row, second column.  
 \* Third character “o” is printed on the first row, third column.  
 \* Fourth character “g” is printed on the second row, fourth column.  
 \* Fifth character “l” is printed on the third row, fifth column.  
 \* Sixth character “e” is printed on the second row, sixth column.  
 \* Seventh character “~” is printed on the first row, seventh column.  
 \* “~” is printed as seventh character in the input is a space.  
 \*  
 \* This process goes on to the last character of the input string.  
 \*  
 \* Output: String will be printed sinusoidally. Format is:  
 \* → There will be 3 rows  
 \* → Print ~ for spaces  
 \* → First character is printed in 1st column of 3rd row  
 \* → Second character is printed in 2nd column of 2nd row  
 \* → Third character is printed in 3rd column of 1st row  
 \* → Fourth character is printed in 4th column of 2nd row  
 \* → Fifth character is printed in 5nd column of 3rd row  
 \* → Sixth character is printed in 6th column of 2nd row  
 \* → This process goes on…  
 \*  
 \* Constraints:  
 \*  
 \* String consisting of alphanumeric characters and spaces  
 \* 3 <= |s| <= 10^5  
 \*  
 \* → Print ~ for spaces  
 \*  
 \* Solution:  
 \*  
 \* There are few observations:  
 \* → i’th character of the string is placed in i’th column of a row.  
 \* → Character at index 0 of string is placed in 3rd row. Then 4th, 8th and go on.  
 \* → Character at index 1 of string is placed in 2nd row. Then 3rd, 5th and go on.  
 \* → Character at index 2 of string is placed in 1st row. Then 6th, 10th and go on.  
 \* So, we can construct 3 strings representing 3 rows using the above information.  
 \*  
 \* Time Complexity:  
 \* O(|s|).  
 \* As we are traversing all the characters of the string,  
 \* so time complexity is O(|s|).  
 \*  
 \* Auxiliary Space:  
 \* O(|s|).  
 \* To store the three rows we use a 2D array with three rows and |s| columns.  
 \* Hence auxiliary space required is O(3\*|s|) = O(|s|)  
 \*  
 \* Space Complexity:  
 \* O(|s|).  
 \*  
 \* Approach:  
 \*  
 \* 1) For 0th row, it places character at col index 2, 6, 10, ... so that is +4  
 \* 2) For 1st row, it places character at col index 1, 3, 5,... so that is +2  
 \* 3) For 2nd row, it places character at col index 0, 4, 8, ... so that is +4  
 \* 4) Fill the first row, second row and third row.  
 \*  
 \* resources/PrintStringSinusoidally.jpg*

# Problem 4:

**Boggle Solver**

**Problem Statement:**

You are given a dictionary set ***dictionary*** that contains ***dictionaryCount*** distinct words and a matrix ***mat*** of size ***n\*m***.

Your task is to find all possible words that can be formed by a sequence of adjacent characters in the matrix ***mat***.

Note that we can move to any of 8 adjacent characters, but a word should not have multiple instances of the same cell of the matrix.

Note: Same dictionary word can be found in the matrix multiple times. We only need to check the existence of the dictionary word in the matrix. Hence, for multiple existences for the same word only add it once in the list of all found words.

**Output Format:**

Return array of strings containing all possible words from the ***dictionary*** that could be found in the matrix ***mat***.

If dictionary = [ ‘’hat”, “world” ] and mat [ “aaa”, “hat”, “ccc”], then the corresponding custom input will be:

2

hat

world

3

3

aaa

hat

ccc

**Output Format:**

Print all the dictionary words found in the matrix mat in a separate line.

For the above-provided custom input only one word is found and hence the custom output looks like:

hat

**Constraints:**

1 <= ***dictionaryCount*** <= 1000

* 1<= ***n\*m*** <= 100000
* 1<= length(words in ***dictionary***) <= 100
* All the characters in ***mat*** and words in ***dictionary*** are lower case English alphabets.

**Sample Test Case:**

***dictionary*** = [ “bst” , “heap” , “tree”]

***mat*** = [ “bsh”  , ”tee” , ”arh” ]

**Sample Output:**

Function returns the list result = [ “bst” , “tree” ]

**Explanation:**

The input matrix is represented below:

bsh

tee

arh

Presence of “bst” is marked bold in the below representation:

(0,0) -> (0,1) -> (1,0)

**bs**h

**t**ee

arh

Presence of “tree” is marked bold in the below representation:

(1,0) -> (2,1) -> (1,1) -> (1,2)

bsh

**tee**

a**r**h

# Solution

**1) brute\_force\_solution:**

In this approach, we, first of all, insert all the words from the ***dictionary*** into a hash map for a linear time lookup operation (linear over length of string because it takes linear time to calculate hash of a string).

Now, we iterate over all the cells of the matrix ***mat*** and assume each cell as the first character of the our word and recursively build all the possible words by visiting all its neighboring cells and each we visit its neighbor we keep building the word by appending the neighbor’s cell char at the end of our current word.

Also, each time we build a word we make a lookup in our hash map. If the current state of the word exists in the hash map then we found it in the ***mat*** and we add it to the found words set. Also, once we found a word we remove it from the HashMap so as to avoid its duplicate match from other words in the ***mat***. We repeat this process for all the cells in the matrix ***mat*** and keep accumulating all the found words in a common container and in the end return this container.

Consider the below example

***dictionary*** = [ “bst” , “abs” , “tab” ]

***mat*** = [ “ast” , “bxr” ]

So, our matrix looks like

a s t

b x r

Now for each cell of the matrix we keep generating all the possible words with max length equal to the max length of word in dictionary.

So, we start from ***mat[0][0***] and keep generating all the words in the below manner and at each generation we check its existence in the dictionary. The generated words for mat[0][0] are:

a , as , ast ,  asx , asr , asb, abx , abs.

Here we find “abs” in our dictionary so, we add it in our found words container and remove it from the dictionary hashmap.

In similar fashion we generate all possible words considering all other cells of ***mat*** as the first character of the word and then keep checking their existence in our dictionary hash map.

**Time Complexity (assuming that input arguments are already given and excluding time used in declaration of output):**

***O(max\_length\_of\_string\*(n\*m)\*7^(max\_length\_of\_string))*** where ***n*** denotes the number of strings in given array ***mat*** , ***m*** denotes the length of a string of given array ***mat*** and ***max\_length\_of\_string*** is the maximum length of the dictionary word.

Now, consider a cell (i,j) as the first character when we are building our word. Now for first move we can move to 8 directions from current cell say (i-1,j) , (i+1,j) , (i,j-1) , (i,j+1) (i-1,j+1) , (i+1,j-1),(i+1,j+1),(i-1,j-1). Now for next moves we only have 7 directions as one direction of the 8 possible direction will be the previous visited cell. So, from this point on we will be having 7 possible directions to visit for the current cell. So, we can now come-up with a upper bound that to form all possible ***max\_length\_of\_string*** length words assuming cell (i,j) is the first character it will take ***O(8 \* 7^(max\_length\_of\_string-1)) ~ O(7 ^ max\_length\_of\_string)*** time***.*** Therefore, we perform the same operation for all the m\*n cells in the matrix mat. Now for words that we form from our boggle matrix we do a lookup in our hashmap to check if the current word exists in the dictionary. This lookup takes ***O(max\_length\_of\_string)*** time. Hence, the total time complexity will be ***O(max\_length\_of\_string\*m\*n\*(7^max\_length\_of\_string)).***

**Time Complexity:**

***O(max\_length\_of\_string \* ( dictionaryCount + \*m\*n\*(7^max\_length\_of\_string) )*** where ***n*** denotes the number of strings in given array ***mat*** , ***m*** denotes the length of a string of given array ***mat*** , ***max\_length\_of\_string*** is the maximum length of the dictionary word and ***dictionaryCount*** denotes the number of words in dictionary.

The input time is the time taken to read the matrix and the dictionary i.e. ***O(n\*m + dictionaryCount \* max\_length\_of\_string).***

The output time is the time taken to print the found words in the dictionary which in worst case is ***O(dictionaryCount\*max\_length\_of\_string).*** Hence, the total time complexity is the sum of input time + function time + output time i.e. ***O(n\*m + dictionaryCount \* max\_length\_of\_string) +*** ***O(max\_length\_of\_string\*m\*n\*(7^max\_length\_of\_string)) + O(dictionaryCount\*max\_length\_of\_string)*** ~ ***O(max\_length\_of\_string \* ( dictionaryCount + \*m\*n\*(7^max\_length\_of\_string) )***

**Auxiliary Space Used:**

***O(dictionaryCount\*max\_length\_of\_string) + O(n\*m)*** where ***dictionaryCount*** is the number of string in given array ***dictionary***, ***max\_length\_of\_string*** is the max length of dictionary string and ***n*** denotes the number of strings in given array ***mat*** and ***m*** denotes the length of a string of given array ***mat***.

Hash Map consumes O(dictionaryCount\*max\_length\_of\_string) space same as the input dictionary. While word building traversals we will be also maintaining a 2D visited matrix that tracks the visited cells so as we do not visit it again. This 2D visited matrix also consumes O(n\*m) space. Recursion stack as we calling the function recursively for any given index (i, j) it can be O(n\*m).

Hence total auxiliary space used will be ***O(dictionaryCount\*max\_length\_of\_string) + O(n\*m).***

**Space Complexity:**

***O(dictionaryCount \* max\_length\_of\_string) + O(n\*m)*** where ***dictionaryCount*** is the number of string in given array ***dictionary***, ***max\_length\_of\_string*** is the maximum length of strings in array dictionary and ***n*** denotes the number of strings in given array ***mat*** and ***m*** denotes the length of a string of given array ***mat***.

For storing input, we are storing ***dictionaryCount*** number of string of maximum possible length ***max\_length\_of\_string*** and ***n*** strings of length ***m*** each and as auxiliary space used is ***O(dictionaryCount \* max\_length\_of\_string) + O(n\*m).*** Hence total complexity will be ***O(dictionaryCount \* max\_length\_of\_string) + O(n\*m).***

**2) optimal\_solution:**

In our previous brute force approach we went building our word string cluelessly and each time we build our string any further we made a lookup in our hash map to check its existence.

In this approach instead of going blindly in all eight directions from our current cell, we will only visit that neighbor that assures that the word with the current prefix exists in the dictionary. Using this we will prune a lot of branches in our word building traversal.

Now a few questions/thoughts:

1. Which DS to choose to store dictionary words so as it not only gives fast lookups but also suggests the next character to look in the matrix for a given prefix word?

* We will be choosing Trie as our DS to store all the words from the dictionary. Trie offers same lookup time as that of a Hash Map(linear time to calculate hash key) and also suggests the next character for a prefix so as the word formed using that char exists in the trie. This feature of the trie is used in autocomplete feature and is widely used in search engines to auto-complete your search queries.

1. Which traversal algorithm to choose for building word(DFS/BFS)?

* The optimal solution for this problem also demands the right choice for the traversal algorithm. In case we use a BFS traversal we will be building valid words but we will be growing our search branches horizontally which leads to delay in the pruning of branches and hence will increase our search space and will affect the time complexity. Whereas in case of DFS we go till depth and we are sure that for every depth we step in we are on right child as guided by the trie data structure and hence, we will reach our goal word quickly.

Now we will use the same approach as used in the brute force solution. We will iterate over all cells of the matrix mat and for each cell we will do a DFS traversal on the matrix assuming that cell as the first character of the word but this time we will be using our trie to guide so as we only land on valid neighbours and increase the chance of finding the dictionary word.

1. Bonus Optimization Catch

* Every time we find a word we simply remove it from the trie. This will ensure that the trie does not give a suggestion of the words that are already found in some previous traversals and hence it will prune some more DFS branches for us.

Consider the below example

***dictionary*** = [ “bst” , “abs” , “tab” ]

***mat*** = [ “ast” , “bxt” ]

So, our matrix looks like

a s t

b x r

Now we iterate on all cells of the above matrix and consider the character as the first character of the word and start building our target word. Here unlike the brute force we won’t build all the possible arrangements of words but will only build those words which are prefix to any of the dictionary word.

So, here we start from ***mat[0][0]***

1. First iteration

***word = “a”*** (prefix of “abs”), now we check if it is in dictionary

1. Second iteration , we visit all neighbours and only append that neighbour characters in our word that trie permits( so the new word formed is still a prefix of words in dictionary)

***word = “ab***” (prefix of “abs”), now we check if it is in dictionary

1. Third iteration , again we repeat the same process

***word = “abs”*** (prefix of “abs”), this exists in the dictionary and hence, we add this in our found words container and remove the same word from the trie.

We repeat the same process for all other cells of the matrix.

**Time Complexity (assuming that input arguments are already given and excluding time used in declaration of output):**

***O(n\*m\*7^(max\_length\_of\_string)) + O(dictionaryCount\*max\_length\_of\_string)*** where ***dictionaryCount*** is the number of string in given array ***dictionary***, ***max\_length\_of\_string*** is the maximum length of strings in array dictionary and ***n*** denotes the number of strings in given array ***mat*** , ***m*** denotes the length of a string of given array ***mat*** and max\_length\_of\_string denotes the maximum length of the dictionary word.

As we are storing each string of array dictionary into trie, it will take ***O(dictionaryCount\*max\_length\_of\_string)*** as to store a string in trie it will take **O(*max\_length\_of\_string*)**.

The time complexity of the DFS word building step for this approach will also be the same as the brute force approach for worst case(kindly refer to the brute force time complexity section). As we are still doing the same DFS traversal as in the brute force approach but with some intelligent choices while choosing the direction from the current cell, so as we form the target word more quickly. But for worst case we will end up forming all possible words even after the guiding given by the trie.

Consider below example :

When our 2D matrix is of size 5\*10 and looks like below:

***mat*** = [

“aaaaaaaaaa”,

“aaaaaaaaaa”,

“aaaaaaaaaa”,

“aaaaaa**xxxx”,**

“aaaaaa**x**bcd”

]

And now consider our dictionary as

***dictionary*** = [ “aaaaaaaaab” , “aaaaaaaaac”, “aaaaaaaaad”]

As it is evident that the letter ‘b’,’c’ and ‘d’ and being shielded by the cover of ‘x’ layer.

Hence and unfortunately total time complexity  still will be ***O(n\*m\*7^(max\_length\_of\_string)) + O(dictionaryCount\*max\_length\_of\_string),*** but this is only in worst case. In average and ideal cases this approach performs much better.

**Time Complexity:**

***O(n\*m\*7^(max\_length\_of\_string) + dictionaryCount\*max\_length\_of\_string))*** where ***dictionaryCount*** is the number of string in given array ***dictionary***, ***max\_length\_of\_string*** is the maximum length of strings in array dictionary and ***n*** denotes the number of strings in given array ***mat*** , ***m*** denotes the length of a string of given array ***mat***

The input time is the time taken to read the matrix and the dictionary i.e. ***O(n\*m + dictionaryCount \* max\_length\_of\_string).***

The output time is the time taken to print the found words in the dictionary which in worst case is ***O(dictionaryCount\*max\_length\_of\_string).*** Hence, the total time complexity is the sum of input time + function time + output time i.e. ***O(n\*m + dictionaryCount \* max\_length\_of\_string) + O(n\*m\*7^(max\_length\_of\_string)) + O(dictionaryCount\*max\_length\_of\_string) + O(dictionaryCount\*max\_length\_of\_string) ~***

***O(n\*m\*7^(max\_length\_of\_string) + dictionaryCount\*max\_length\_of\_string))***

**Auxiliary Space Used:**

***O(n\*m) + O(dictionaryCount\*max\_length\_of\_string)*** where ***dictionaryCount*** is the number of string in given array ***dictionary***, ***max\_length\_of\_string*** is the maximum length of strings in array dictionary and ***n*** denotes the number of strings in given array ***mat*** and ***m*** denotes the length of a string of given array ***mat***.

Trie consumes ***O(dictionaryCount\*max\_length\_of\_string)*** space to store ***dictionaryCount*** of strings in worst case. While word building traversals we will be also maintaining a 2D visited matrix that tracks the visited cells so as we don’t visit it again. This 2D visited matrix also consumes O(***n\*m***) space and the recursive stack would take O(***max\_length\_of\_string***) space. So, overall auxiliary space is ***O(n\*m) + O(dictionaryCount\*max\_length\_of\_string).***

**Space Complexity:**

***O(n\*m) + O(dictionaryCount\*max\_length\_of\_string)*** where ***dictionaryCount*** is the number of string in given array ***dictionary***, ***max\_length\_of\_string*** is the maximum length of strings in array dictionary and ***n*** denotes the number of strings in given array ***mat*** and ***m*** denotes the length of a string of given array ***mat***.

For storing input, we are storing ***dictionaryCount*** number of string of maximum possible length ***max\_length\_of\_string*** and ***n*** strings of length ***m*** each and as auxiliary space used is ***O(n\*m) + O(dictionaryCount\*max\_length\_of\_string)*** hence total complexity will be ***O(dictionaryCount\*max\_length\_of\_string) + O(n\*m).***