# Problem 1:

**Regex Matcher**

Given a *text* string containing characters only from lowercase alphabetic characters and a *pattern* string containing characters only from lowercase alphabetic characters and two other special characters '.' and '\*'.

Your task is to implement a pattern matching algorithm that returns true if *pattern* is matched with *text* otherwise returns false. The matching must be exact, not partial.

Explanation of the special characters:

1) '.' - Matches a single character from lowercase alphabetic characters.

2) '\*' - Matches zero or more preceding character. It is guaranteed that '\*' will have one preceding character which can be any lowercase alphabetic character or special character '.'. But '\*' will never be the preceding character of '\*'. (That means "\*\*" will never occur in the *pattern* string.)

'.' = "a", "b", "c", ... , "z".

a\* = "a","aa","aaa","aaaa",... or empty string("").

ab\* = "a", "ab", "abb", "abbb", "abbbb",...

**Example One**

Input: *text* = "abbbc" and *pattern* = "ab\*c"

Output: true

Given *pattern* string can match:

"ac", "abc", "abbc", "abbbc", "abbbbc", ...

**Example Two**

Input: *text* = "abcdefg" and *pattern* = "a.c.\*.\*gg\*"

Output: true

".\*" in *pattern* can match  "", ".", "..", "...", "....", ...

"g\*" in *pattern* can match "", "g", "gg", "ggg", "gggg", ...

Now, consider:

    '.' at position 2 as 'b',

    '.\*'  at position {4,5} as "...",

    '.\*' at position {6,7} as "" and

    [g\*] at position {8,9} as "".

So, "a.c.\*gg\*" = "abc...g" where we can write "..." as "def". Hence, both matches.

**Example Three**

Input:

*text* = "abc" and *pattern* = ".ab\*.."

Output: false

If we take 'b\*' as "" then also, length of the *pattern* will be 4 (".a.."). But, given *text*'s length is only 3. Hence, they can not match.

**Notes**

Input Parameters: There are two arguments, first one is string denoting *text* and second one is string denoting *pattern*.

Output: Return boolean, true if *text* and *pattern* matches exactly, otherwise return false.

Constraints:

* 0 <= *text* length, *pattern* length <= 1000
* *Text* string containing characters only from lowercase alphabetic characters.
* *Pattern* string containing characters only from lowercase alphabetic characters and two other special characters '.' and '\*'
* In *pattern* string, It is guaranteed that '\*' will have one preceding character which can be any lowercase alphabetic character or special character '.'. But '\*' will never be the preceding character of '\*'.

# Problem 2:

**KMP**

Given a text string *t* of length *n* and a pattern string *p* of length *m*, return start indices of all occurrences of *p* in *t*.

You have to answer *q* such queries.

**Example One**

Input:

*t* = "Ourbusinessisourbusinessnoneofyourbusiness"

*q* = 3 (t will be same for all 3 queries)

*p* in 1st query = "business"

*p* in 2nd query = "our"

*p* in 3rd query = "daisy"

Output:

3

16

34

13

31

-1

**Example Two**

Input:

*t* = "IfyouthinkyouthinktoomuchthenyoumightbewrongThinkaboutit"

*q* = 4 (t will be same for all 4 queries)

*p* in 1st query = "aaaa"

*p* in 2nd query = "think"

*p* in 3rd query = "you"

*p* in 4th query = "be"

Output:

-1

2

5

10

13

29

37

**Notes**

Input Parameters: There are two arguments, *t* and *p*, denoting text string and pattern string respectively.

Output: Return an array of integer *pos*[], where *pos*[i] is the start index of ith occurrence of *p* in t(0-based indexing).

Constraints:

* 1 <= *q* <= 5
* 1 <= *n* <= 2\*10^5
* 1 <= *m* <= 2\*10^5
* *t* and *p* may contain lower case characters, upper case characters and numeric characters.

If there is no occurrence of p in t, then return array *pos* of size one with *pos*[0] = -1.

If there are multiple occurrences of p in t, then return an array of start indices(sorted in increasing order).

If

*t* = "Ourbusinessisourbusinessnoneofyourbusiness",

*q* = 3,

then input should be:

Ourbusinessisourbusinessnoneofyourbusiness

3

business

our

output will be:

3

16

34

13

31

-1

# Solution

Given a text string t of length n and a pattern string p of length m, return start indices of all occurrences of p in t.

**1. brute\_force.java**

A naive approach would be to iterate over all possible substrings of text t having length m and compare it with pattern p.

As there are total (n-m+1) such substrings, each of length m, total no of comparisons would be O(n\*m)

**Time Complexity:**

O(n\*m) for each query

Overall time complexity: O(q\*n\*m)

**Auxiliary Space Used:**

O(1)

**Space Complexity:**

O(n+m)

**2. optimal\_solution.java**

An optimal approach would be as follows:

KMP (Knuth Morris Pratt) Pattern Matching Algorithm

The Naive pattern searching algorithm doesn’t work well in many cases as we slide the pattern by one and compare all characters at each shift until we find a mismatch.

Following are some examples.

t = "CCCCCCCCCCCCCCCCCCCCCCCCCCCCB"

p = "CCCCCB"

t = "CDCDCDECDCDCDECDCDCDE"

p = "CDCDCE"

The basic idea behind KMP’s algorithm is: whenever we detect a mismatch (after one or more matches), we already know some of the characters are going to match in the text of next window. We take advantage of this information to avoid matching the characters that we know will anyway match and thus avoiding redundant comparisons.

Example:

t = "CCCCCDCCCDC"

p = "CCCC"

We compare first window of text t with pattern p: (i.e. starting with 1st char of t)

t = "CCCCCDCCCDC"

p = "CCCC"  [Initial position]

We find a match. This is the same as in the naive algorithm.

In the next step, we compare next window of t with p (i.e. starting with 2nd char of t)

t = "CCCCCDCCCDC"

p =  "CCCC" (Pattern shifted one position)

This is where KMP does optimization over naive. In this second window, we only compare 4th A of pattern with 4th character of current window of text to decide whether current window matches or not. Since we know first three characters will anyway match, we skipped matching first three characters.

How to know how many characters to be skipped?

To know this, we pre-process pattern and prepare an integer array lps[] that tells us the count of characters to be skipped. To be precise, lps[i] = x shows that if first un-matched character is found at (i+1)th char of p, then for next comparison, you can skip the first x chars of p.

Preprocessing:

KMP algorithm preprocesses p[] and constructs an auxiliary array lps[] of size m (same as size of pattern) which is used to skip characters while matching.

Name lps indicates the longest proper prefix which is also suffix. A proper prefix is prefix with the whole string not allowed.

For example, prefixes of “ABC” are “”, “A”, “AB” and “ABC”. Proper prefixes are “”, “A” and “AB”.

Suffixes of the string are “”, “C”, “BC” and “ABC”.

For each substring p[0..i] where i = 0 to m-1, lps[i] stores length of the maximum matching proper prefix which is also a suffix of the substring pat[0..i].

  lps[i] = the longest proper prefix of pat[0..i] which is also a suffix of pat[0..i].

Note : lps[i] could also be defined as the longest prefix which is also a proper suffix. We need to use "proper" at least one place to make sure that the whole string is not considered.

Examples of lps[] construction:

For the pattern “CCCC”,

lps[] is [0, 1, 2, 3]

For the pattern “CDEFG”,

lps[] is [0, 0, 0, 0, 0]

For the pCttern “CCDCCECCDCC”,

lps[] is [0, 1, 0, 1, 2, 0, 1, 2, 3, 4, 5]

For the pCttern “CCCDCCCCCD”,

lps[] is [0, 1, 2, 0, 1, 2, 3, 3, 3, 4]

For the pCttern “CCCDCCC”,

lps[] is [0, 1, 2, 0, 1, 2, 3]

To build lps array, for lps[i], we keep track of the length of the longest proper prefix suffix value for the previous index(let's call it 'len'). We initialize lps[0] and len as 0. If p[len] and p[i] matches, we increment len by 1 and assign the incremented value to lps[i]. If p[i] and p[len] do not match and len is not 0, we update len to lps[len-1].

See the implementation for better understanding.

We use a value from lps[] to decide the next characters to be matched. As mentioned before, the idea is to not match a character that we know will anyway match.

How to use lps[] to know a number of characters to be skipped?

We start comparison of p[j] with j = 0 with characters of the current window of text.

We keep matching characters t[i] and p[j] and keep incrementing i and j while pat[j] and txt[i] keep matching.

When we see a mismatch, we know that characters p[0,1,..,j-1] match with t[i-j,..,i-1]

(Note that j starts with 0 and increments only when there is a match).

We also know that lps[j-1] is count of characters of p[0,1,..,j-1] that are both proper prefix and suffix.

From above two points, we can conclude that we do not need to match these lps[j-1] characters with txt[i-j,..,i-1] because we know that these characters will anyway match.

See the implementation for better understanding.

**Time Complexity:**

O(n) for each query

Over all time complexity: O(q\*n)

**Auxiliary Space Used:**

O(m)

**Space Complexity:**

O(n+m)

# Problem 3

**Longest Substring With Exactly Two Distinct Characters**

Given a string *s* of length *n*, find the length of the longest substring *ss*, that contains exactly two distinct characters.

There will be *t* test cases.

**Example One**

Input:

2

eceba

abcdef

Output:

3

2

In the first case, 'ece' is the largest substring with exactly 2 distinct characters.

In the second case, 'ab' is the largest substring with exactly 2 distinct characters. Also, 'bc', 'cd', 'de', 'ef' can be considered as substring with exactly 2 distinct characters.

**Example Two**

Input:

3

ababababa

e

baabcbab

Output:

9

0

4

In the first case, the whole string 'ababababa' is the largest substring with exactly 2 distinct characters.

In the second case, there is no substring with exactly 2 distinct characters.

In the third case, 'baab' is the largest substring with exactly 2 distinct characters.

**Notes**

Input Parameters: There is only one argument *s*, denoting the input string.

Output: Return an integer *len*, denoting length of *ss*.

(If there are no such substrings, then return 0)

Constraints:

* 1 <= *t* <= 80
* 1 <= *n* <= 10^5
* *s* may contain upper case alphabets, lower case alphabets and numerical values.

**Custom Input**

Input Format: The first line of the input should contain an integer *t*, denoting no. of test cases.

In the next *t* lines, ith line should contain a string *si*, denoting an input string *s* for ith test case.

If *t* = 3, *s* for 1st test case = “ababababa”, *s* for 2nd test case = “e” and *s* for 3rd test case = “baabcbab”, then input should be:

3

ababababa

e

baabcbab

Output Format: There will be t lines for output, where ith line contains an integer *leni*, denoting resultant value *len* for ith test case. For input *t* = 3, *s* for 1st test case = “ababababa”, *s* for 2nd test case = “e” and *s* for 3rd test case = “baabcbab”, output will be:

9

0

4

# Solution

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

A naive approach would be to iterate over all possible substrings ss of input string s, check if it is a valid (i.e. contains exactly two distinct characters) substring or not. Maximum value of length out of all valid substrings is the required output.

While iterating over all substrings, we will iterate in such a manner that first all the substrings which starts from 0th index are covered, then the ones which starts from 1st index are covered, then ones from 2nd index, 3rd index and so on. And, while iterating over all substrings starting at ith index, we will break iteration while we hit on the third distinct character found at index j>i.

**Time Complexity:**

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

In the worst case, where we have only two distinct characters in the whole string, all substrings are valid. So, using this approach, code will iterate over all n^2 substrings.

**Auxiliary Space Used:**

O(1).

We are not storing anything extra.

**Space Complexity:**

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

Input of function will take O(n) to store string s and as auxiliary space used is O(1).

O(n) + O(1) → O(n).

**2) optimal\_solution.java**

An optimal approach would be as follows:

Let say, for current substring, starting point is i and j (for current value of i) is the index where there is first third distinct character, considering substring starting at i.

Now, using the two pointer approach, move i forward (i.e i=i+1) until there are only two distinct characters in window [i,j] representing substring. Maintain a max variable 'res', which will be updated each time you find a valid substring while iterating string using two pointer approach. For maintaining frequency of characters in the current window [i,j], we can use HashMap or similar data structure which allows O(1) amortized time complexities for lookup and addition.

**Time Complexity:**

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

As in this two pointer approach, none of the two pointers ever moves backward (i.e. to a smaller value than its current value), the complete string will be iterated twice only. Twice iteration of string will take O(n).

**Auxiliary Space Used:**

O(1).

We are just maintaining a frequency map for at max three characters at any time hence O(1).

**Space Complexity:**

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

Input of function will take O(n) to store string s and as auxiliary space used is O(1).

O(n) + O(1) → O(n).

# Problem 4:

**Join Words To Make A Palindrome**

Given a list of strings *words*, of size *n*, check if there is any pair of words that can be joined (in any order) to form a palindrome then return the pair of words forming palindrome.

**Example One**

Input: *words* = [ “bat”, “tab”, “zebra” ]

Output: *result* = [ “bat”, “tab” ]

As “bat” + “tab” = “battab”, which is a palindrome.

**Example Two**

Input: *words* = [ “ant”, “dog”, “monkey” ]

Output: *result* = [ “NOTFOUND”, “DNUOFTON” ]

As for each 6 combinations of string of *words*, there is no single generated word which is a palindrome hence *result* list will be [ “NOTFOUND”, “DNUOFTON” ].

**Notes**

Input Format: Only argument for function, list of strings *words*.

Output:

Return a pair of words (i.e. list of string *result* of size 2 such that *result*[0] + *result*[1] is a palindrome).

In case of multiple answers return any one of them.

In case of no answer return list [“NOTFOUND”, “DNUOFTON”].

Constraints:

* Length *l* for each word of words list, 1<= *l* <= 30.
* Size of list words n, 2 <= *n* <= 20000.
* Characters for each word can be from [a-z], [A-Z], [0-9].

# Solution

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

A naive approach would be to iterate over all ordered pairs of words from list *words*, i.e. (*words*[i], *words*[j]) such that i != j, 0<=i<*n*, 0<=j<*n*, check if *words*[i] + *words*[j] is palindrome or not. If we found such a pair of words forming a palindrome then return that pair of words.

**Time Complexity:**

O(*(n^2)\*l*), where *n* is size of list *words* and *l* is the maximum length of words in list *words*.

As there are total 2\*(*n*C2) ordered pair of words, and for each pair, for finding whether that pair is forming palindrome or not will take O(*l*).So, time complexity of this solution will be O(*(n^2)\*l*).

**Auxiliary Space Used:**

O(*l*) where *l* is the maximum length of words in list *words*.

As we are storing *result* list of two words of maximum length *l*.

**Space Complexity:**

O(*n\*l*) where *n* is size of list *words* and *l* is the maximum length of words in list *words*.

Input will take space O(*n\*l*) because we are storing *n* words of list *words* where maximum possible length of word can be *l* and auxiliary space used is O(*l*). So, O(*n\*l*) + O(*l*) -> O(*n\*l*).

# Optimal Solution using Trie

1. Build trie of all words - O(nk)
2. Search reverse of words in trie – O(nk)
3. Build trie of reverse words – O(nk)
4. Search reverse of words in trie – O(nk)

# Problem 5

**Longest Repeated Substring**

Given a string *inputStr* of length *n*, find the longest repeated substring in it.

- Repeated substring is a substring that occurs more than once in the given string.

- If there are multiple such substrings of the same size, then return any one.

- If there are no repeated substrings, then return an empty string.

**Example One**

Input: “aaaa”

Output: “aaa”

aaa is the longest substring which is repeated in aaaa, starting at position 0 and starting at position 1.

**Example Two**

Input: “efabcdhefhabcdiefi”

Output: “abcd”

“abcd” repeats twice and “ef” repeats three times. Of those two repeated substrings, “abcd” is the longer one.

**Example Three**

Input: “abcdefghi”

Output: “”

There are no repeated substrings in “abcdefghi”.

**Notes**

Input Parameters: There is only one argument *inputStr*, denoting input string.

Output: Return a string *lrs*, denoting longest repeated substring.

Constraints:

* 2 <= *n* <= 2\*10^5
* *inputStr* may contain only lowercase characters a-z.

This is purely an exercise in building a Suffix Tree.

Suffix trees are difficult. You'd probably wonder if they really ask those in an interview.

They in fact are rarely asked, which is why we don't cover it in the class. But we've seen them at FB and Uber. In all occasions, it's been asked as a follow up question. Once you code up an N^2 algorithm for the problem on hand, there are a few minutes left, in which time, the interviewer would wonder if you know of Suffix trees. It is NEVER asked to implement one in an interview. That's stupid. If at that time, you do know of suffix trees, then you have a chance to convert that interviewer from a 3 (good) to a 4 (advocate). It suggests you have taken a keen interest in your prep work and by extension, in general CS.

Another reason we include it in the course: it is possibly one of the hardest data structures. Once you have a handle on it, a lot of other things will look easy ;-)

Doing difficult problems like these also has a strong ancillary benefit: it helps you indirectly interview your interviewer/company. You want to work for a team that challenges you; not the team that gives you a free pass.

Don't skimp on it. Take it head on, there are clear benefits.

**Custom Input**

Input Format: There should be only one line, containing *inputStr*, denoting input string.

If *inputStr* = “efabcdhefhabcdiefi”, then input should be:

efabcdhefhabcdiefi

Output Format: There will be only one line, containing a string *lrs*, denoting longest repeated substring.

For input *inputStr* = “efabcdhefhabcdiefi”, output will be:

abcd

# Solution

**1. brute\_force.java**

Naive approach would be as follows:

Iterating over all possible non empty substrings, find the number of occurrences of each substring in inputStr. The longest substring having a number of occurrences greater than 1 is the required substring. As there are total O(n^2) substrings, and for each substring, we are computing its number of occurrences in O(n^2), time complexity of this approach would be O(n^4)

**Time Complexity:**

O(n^4)

**Auxiliary Space Used:**

O(n)

**Space Complexity:**

O(n)

**2. other\_solution.java**

To find the number of occurrences of substring in inputStr here, we can use any linear order pattern matching algorithm, for example KMP. Then, we will be able to compute the number of occurrences for each possible substring in O(n) time complexity. Time complexity of the above mentioned approach using this optimization will be O(n^3).

(Notations:

- lrs => Longest Repeated Substring

- lrslen => length of Longest Repeated Substring

- ith suffix of string str => substring starting at index i and ending at end of string

- lcp => longest common prefix

)

Better approach would be as follows:

Let's say we have lrs of length lrslen, which starts at ith index and jth index of inputStr. That means the longest matching prefix of ith suffix and jth suffix is lrs. As it is lrs, no other pair of distinct suffixes has longer matching prefix.

So, our problem boils down to finding the length of longest matching prefix for all pairs of distinct suffixes and the pair (xth suffix, yth suffix) with largest length of lcp will lead to the solution.

This is a dynamic programming problem.

Let say lcp[i][j] denotes the longest common prefix for a pair of ith suffix and jth suffix. Then,

if inputStr[i] == inputStr[j]

    lcp[i][j] = 1 + lcp[i+1][j+1]

else

    lcp[i][j] = 0

And lrslen = max(lcp[i][j]), where i!=j, 0 <= i,j <= n-1

And let say, (x,y) is a value of (i,j) for which lrslen = lcp[i][j], then inputStr.substring(x,x+lrslen-1) is the required lrs.

As we are finding lcp, in O(1), for all O(n^2) pair of suffixes, time complexity will be O(n^2) and as we need to store entire lcp array of size n\*n, space complexity will be O(n^2).

Space complexity can be optimized to O(n) with a better implementation of this approach. (Exercise for reader :D )

**Time Complexity:**

O(n^2)

**Auxiliary Space Used:**

O(n^2)

**Space Complexity:**

O(n^2)

**3. optimal\_solution.java**

It can be further optimized using Suffix Tree Data Structure.

Please read about what suffix tree is and its construction from:

<https://www.geeksforgeeks.org/ukkonens-suffix-tree-construction-part-1/>

(

- Path length of node here means number of characters in path from root to this node, unless explicitly mentioned other meaning

- Just to be clear, root node is not considered as an internal node

- ith suffix is a suffix having start index i

- ith suffix has suffixIndex = i

- Each node in Suffix Tree constructed here has fields: begin, end, depth, parent, childrens[], suffixIndex, suffixLink

)

As mentioned in the previous approach, our problem of finding the longest repeated substring boils down to finding a pair of suffixes having the longest common prefix.

Rewriting this in context of suffix tree, we need to find node resNode, such that:

- resNode is a maximum path length node such that more than one leaf node in its subtree, i.e. resNode has more than one suffixes ends in resNode's subtree.

That implies resNode is one of the internal nodes.

Now the claim is, the resNode is one of the immediate parent nodes of leaf nodes.

To prove this claim, let's say resNode is not one of the immediate parent nodes of leaf nodes. That implies resNode has at least one internal node iNode as it's child node. Now as iNode is an internal node, it has more than one leaf node in its subtree. Also, as it is a child of resNode, pathLength(iNode)>pathLength(resNode) always since the edge between resNode and iNode must contain at least one character. CONTRADICTION as we found another node iNode in the suffix tree which has more than one leaf in its subtree and it's pathlength is greater that pathlength of resNode. So, our assumption that resNode is not one of the immediate parent nodes of leaf nodes can't be true.

So, we are now ready with all subparts to solve this problem.

Here is the pseudoCode.

lrsLength = 0, lrsSuffixIndex=-1

findLRS(Node node){

    isLeaf = true;

    for (Node childNode : node.childrensList) {

        isLeaf = false;

        findLRS(childNode);

    }

    if (isLeaf) {

        currLength = node.parent.pathLength;

        if (currLength > lrsLength) {

            lrsLength = currLength;

            lrsSuffixIndex = node.suffixIndex;

        }

    }

}

Substring starting at (lrsSuffixIndex) and ending at (lrsSuffixIndex+lrsLength-1) is the longest repeated substring

(There may be multiple lrs in inputStr. Here, we are finding any one.)

As time complexity to construct a suffix tree is O(n), and we are traversing all O(n) nodes of the suffix tree to find lrs, overall time complexity of solution will be O(n).

As there are O(n) nodes in the suffix tree, overall space complexity of this approach will be O(n).

There are minor changes in implementation of this approach. See the solution code file for better understanding of approach and implementation.

**Time Complexity:**

O(n)

**Auxiliary Space Used:**

O(n)

**Space Complexity:**

O(n)

Reference : <https://sites.google.com/site/indy256/algo/suffix_tree>

<https://en.wikipedia.org/wiki/Ukkonen%27s_algorithm>

We note that time and space complexities of suffix tree construction depend on the size of the allowed set of characters, depending on the type of implementation. Ukkonen’s algorithm for suffix tree construction, which we have used here, runs  in O(n) if size of allowed set of characters is constant and O(n\*NO\_OF\_CHARACTERS) without this assumption. As we have a constant size of the set of allowed characters mentioned in this problem, it is O(n) time and space complexity.

Exercise for readers:

- This problem can also be solved using:

1. Suffix array

2. String hashing, binary search

- Extension of this problem:

Find the longest repeated substring for a given string s of length n, which occurs at least k times. Same constraints.

# Problem 6:

**Generate Numeronyms**

Given a string *word* of length *n*, generate all possible numeronyms.

What is a Numeronym?

A numeronym is a word where a number is used to form an abbreviation.

For a given string *word*, a numeronym is a string with few or more contiguous letters between the first letter and last letter of *word* replaced with a number representing the count of letters omitted. Only one set of contiguous letters are replaced by a number.

e.g. “L10n” is called a numeronym of the word “Localization”, where 10 stands for the count of letters between the first letter 'L' and the last letter 'n' in the word.

**Example**

Input: “nailed”

Output: ["n4d", "na3d", "n3ed", "n2led", "na2ed", "nai2d"]

“n4d” is an abbreviation of given string “nailed” where “aile” string letters are omitted and replaced by count of letters, i.e. 4.

“na3d” is an abbreviation of given string “nailed” where “ile” string letters are omitted and replaced by count of letters, i.e. 3.

And so on.

**Notes**

Input Format: Only one argument denoting input string *word*.

Output: Return strings array containing all possible numeronyms of given string *word*.  You don’t need to worry about the order of strings in your output array. For *words* = "aaaaa", arrays ["a2aa", "aa2a", "a3a"], ["a3a", "aa2a", "a2aa"] etc will be considered a valid answer. In case of no possible numeronym string, return empty list.

Constraints:

* String will be composed of characters [a-z], [A-Z], [0-9] only.
* 1 <= n <= 120 where *n* is length of given string *word*.

# Solution

We provided an optimal solution.

**optimal\_solution.java**

For any given string str, length of omitted characters l can be 2 <= l <= n-2 where n is length of string as we can’t omit first and last characters and we need to find a numeronym in which at least 2 contiguous letters were omitted.

So for any given length l, we iterate over all possible positions from where omission of characters can start, find a string of length l from that position and replace that with l (number of omitted characters).

**Time Complexity:**

O(n^3) where n is length of given string str.

As iteration will be in three loops, first over possible lengths then over possible first characters of omitted characters and then to find store newly created numeronym.

**Auxiliary Space Used:**

O(n^3) where n is length of given string str.

Maximum number of possible numeronyms generated can be O(n^2) and length of each will be O(n) hence it takes O(n^3) to store output.

**Space Complexity:**

O(n^3) where n is length of given string str.

It will be equal to auxiliary space as in input we are just reading a single input string of length n which takes O(n).

O(n^3) + O(n) → O(n^3)