# Video Link: <https://youtu.be/A-Xl-__MSYA>

Strings are immutable in java. That means you cannot change it.

If you want to change the string you use StringBuffer(synchronized) or StringBuilder.

# Problem 1: Reverse a string

1. Convert string to char array.
2. Reverse char array by swapping first and the last char.
3. Convert char array to string. String.valueOf(charArray).

Time Complexity: O(n)

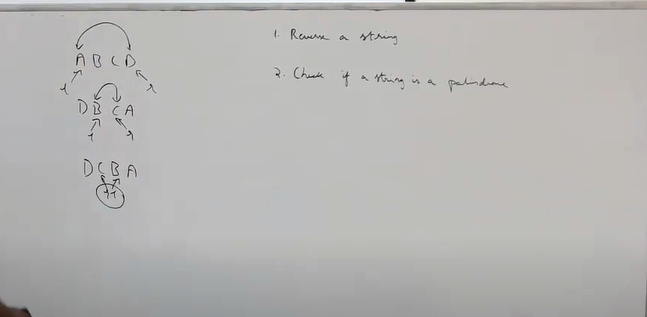
Space Complexity: O(n)

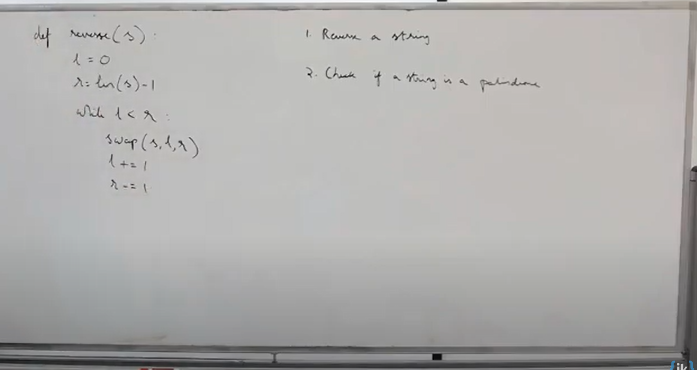
# Problem 2: Check whether string is palindrome or not.

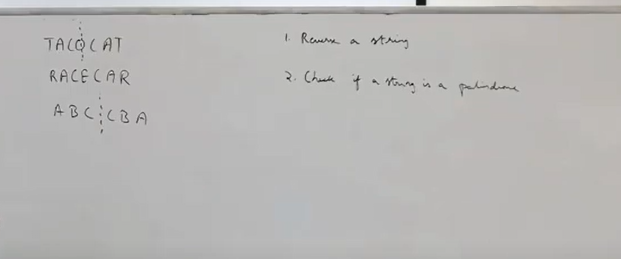
1. Two pointer one from start and other at end.
2. If they are not same, then return false.
3. At the end return true.

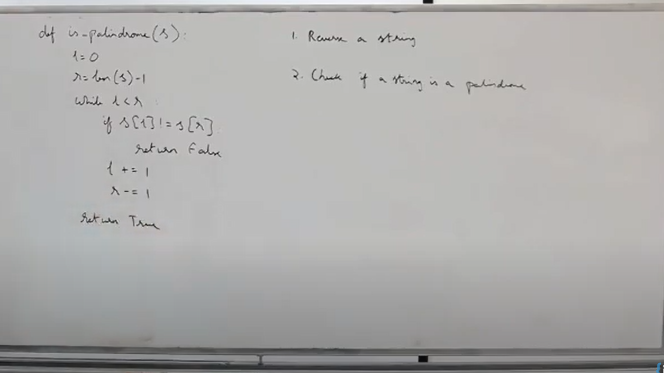
Time Complexity: O(n)

Space Complexity: O(1)

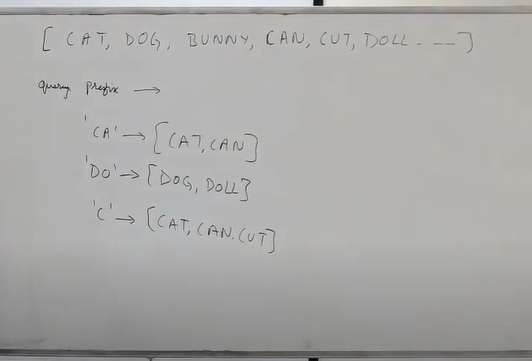








# Prefix Tree or Tries



Imagine you are given an array of words or dictionary of words.

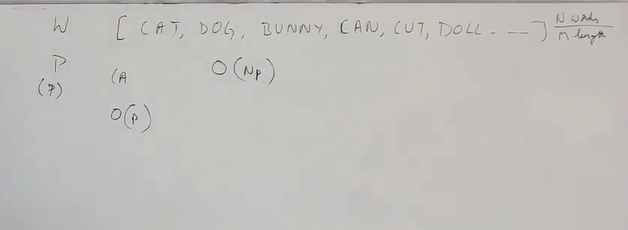
You are given query prefix and you have to find all the words with that query prefix.

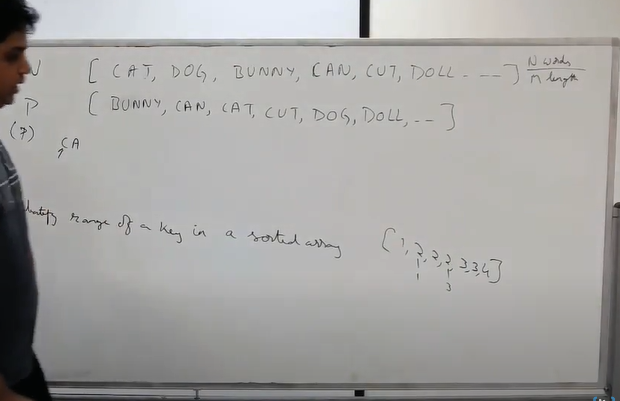
Above is the brute force approach.

Length of query prefix = p

N words

Time Complexity: O(np)





Sort the input.

# Problem: Identify range of a key in a sorted array.

*/\*\*  
 \* Find First and Last Position of Element in Sorted Array  
 \* Given an array of integers nums sorted in ascending order, find the  
 \* starting and ending position of a given target value.  
 \*  
 \* Your algorithm's runtime complexity must be in the order of O(log n).  
 \*  
 \* If the target is not found in the array, return [-1, -1].  
 \*  
 \* Example 1:  
 \*  
 \* Input: nums = [5,7,7,8,8,10], target = 8  
 \* Output: [3,4]  
 \* Example 2:  
 \*  
 \* Input: nums = [5,7,7,8,8,10], target = 6  
 \* Output: [-1,-1]  
 \*  
 \* Approach: Modified Binary Search - 2 pass  
 \*  
 \* 1) We don't exit out when array[mid] == target, we also make sure it's the first key.  
 \* 2) We don't exit out when array[mid] == target, we also make sure it's the last key.  
 \* 3) The base condition differs in both binary search.  
 \* 4) For first binary search, to find the starting index, if target <= array[mid], go to left  
 \* 5) For last binary search, to find the ending index, if target >= array[mid], go to right.  
 \*  
 \* Time Complexity: O(logn)  
 \* Space Complexity: O(1)  
 \*  
 \* resources/FirstAndLastPositionInSortedArray1.jpg  
 \* resources/FirstAndLastPositionInSortedArray2.jpg  
 \*/*

[bunny, can, cat, cut, dog, doll]

1. Sort the array
2. Find the range of all the words starting with ‘c’ using modified binary search – 2 pass.
3. Again do the binary search on the range for second character ‘a’.

Key = ‘ca’

Time Complexity:

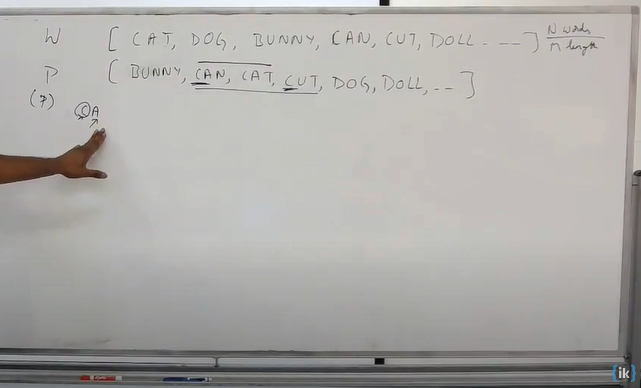
Sort the array = O(nmlogn)

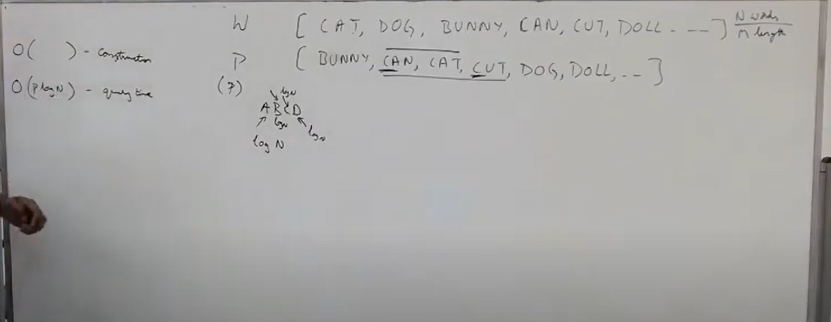
Partition the array of size n takes logn time.

Merge takes nm time where m = length of each word.

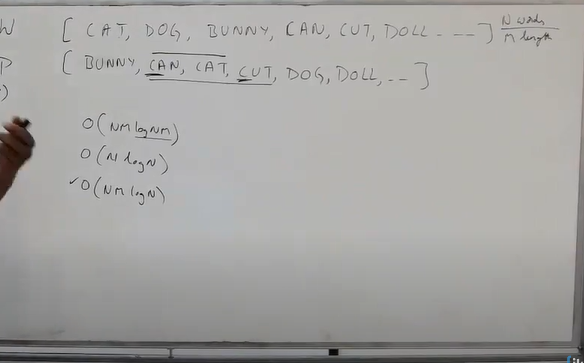
Query time = O(plogn) where p = length of key.

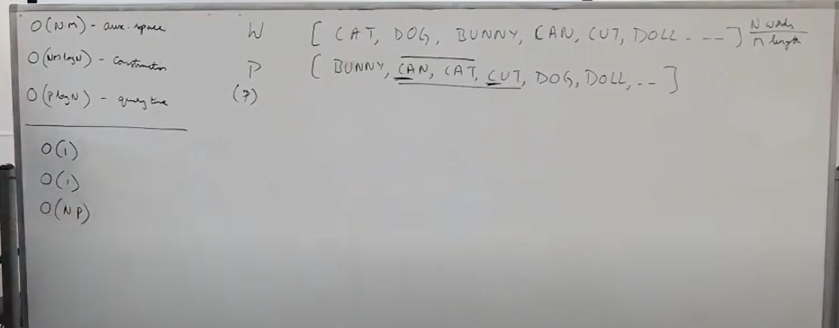
Total time complexity = O(nmlogn + plogn)





Sort the dictionary

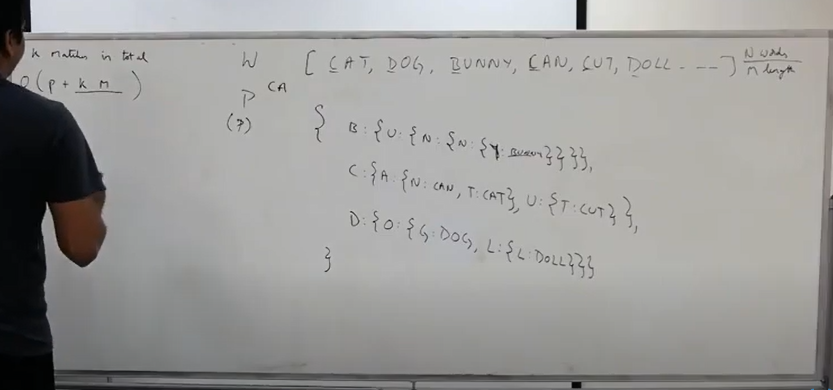




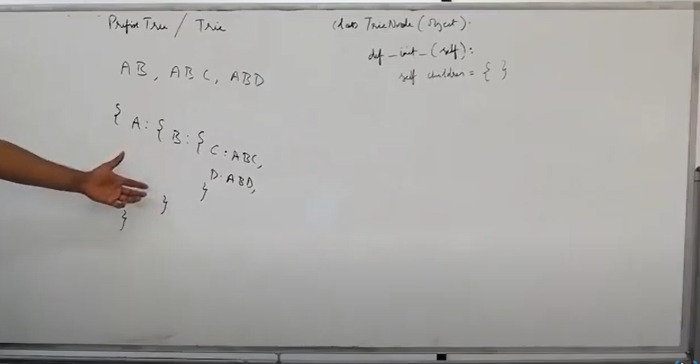
Space Complexity for sorting = O(NM) because we are changing the given array.

Modified binary search versus brute force comparison.

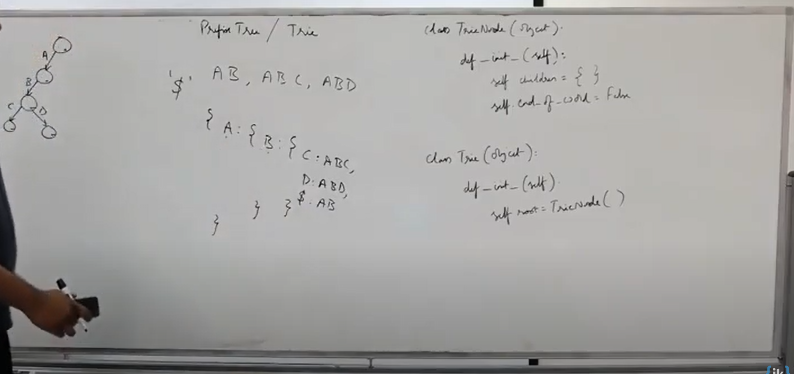
# Trie or prefix tree.



# Search word “AB” in below Trie

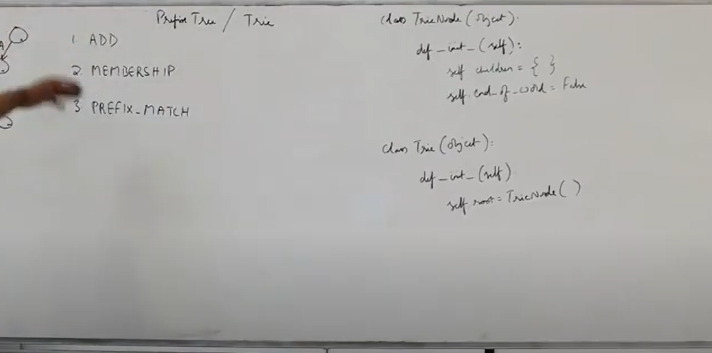


endOfWord flag to denote it’s a word.

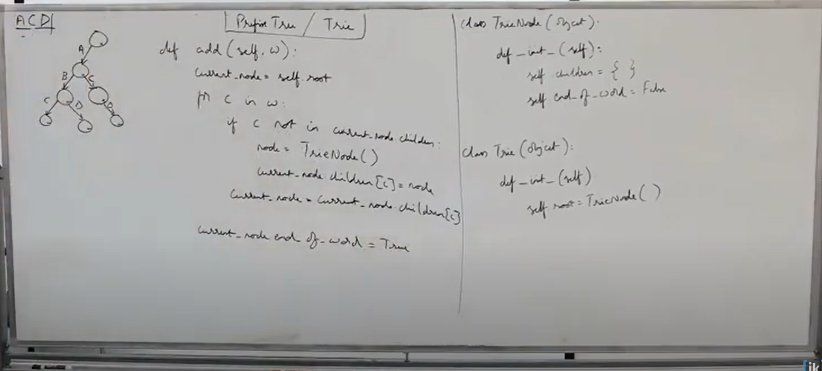


The operations on Tries are

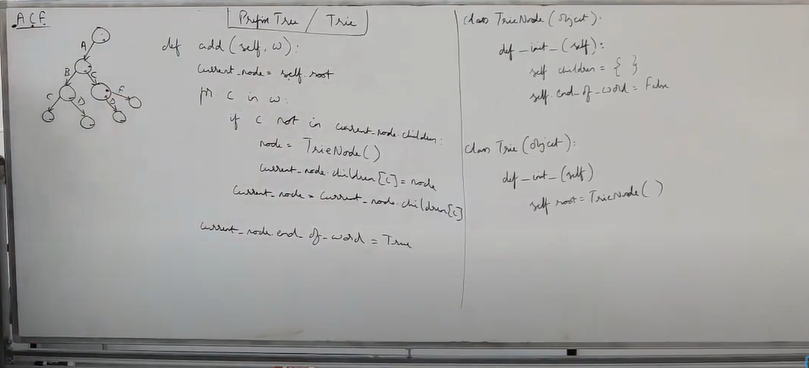
1. Insert
2. Search for a word
3. Search all the words with given prefix



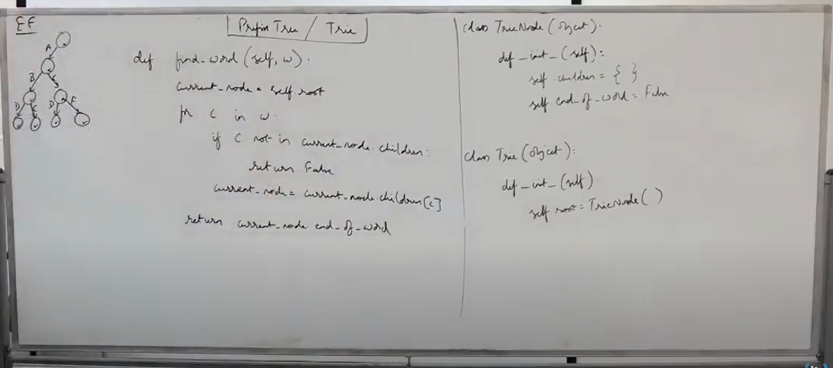
# Insert



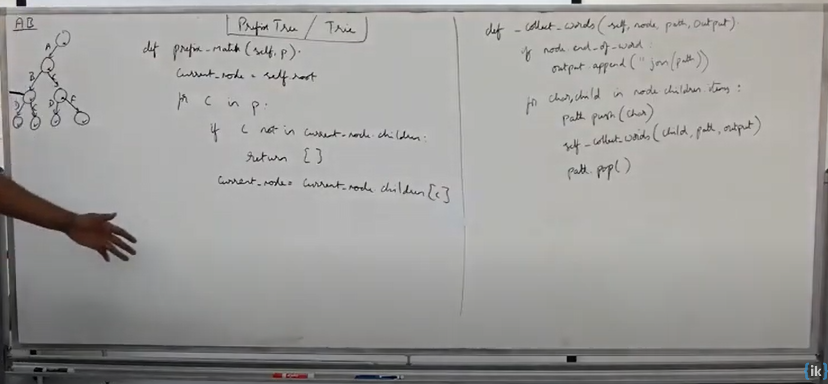
# Insert “ACF”

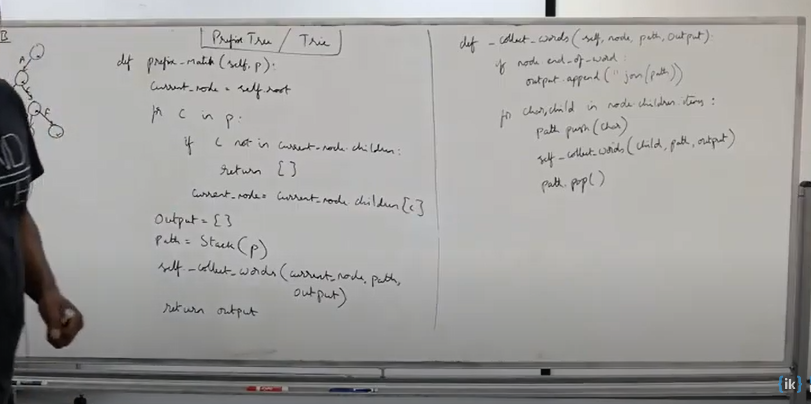


# Search word in tries



# Prefix match – return all the words matching prefix

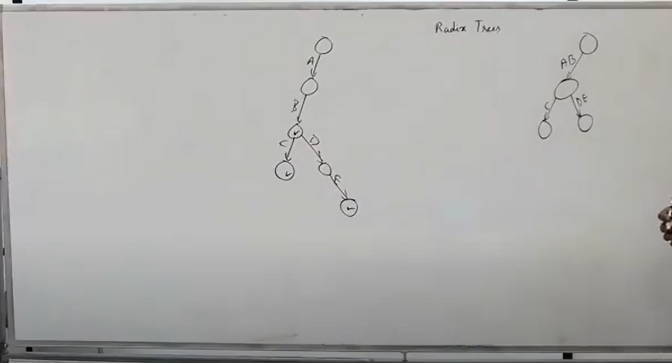




DFS after the prefix node to find all the matching words.

# Radix Tree

A chain of single children can be merged into single node.

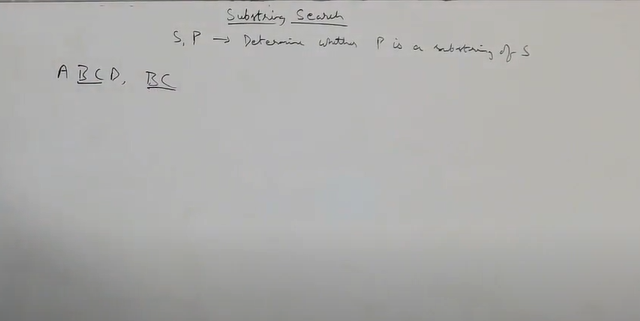




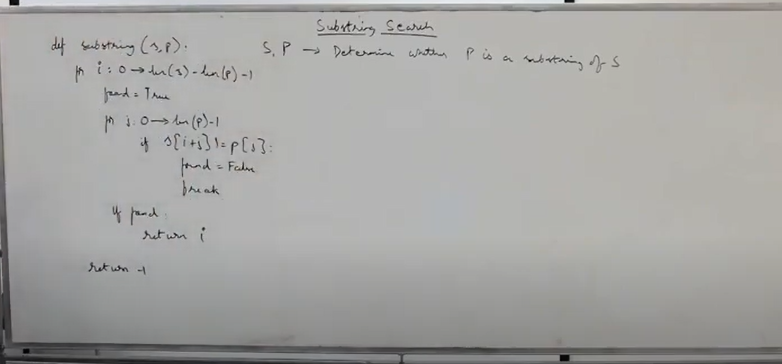
The time complexity of Radix Tree and Tries remaining the same.

It just compresses the node and save space.

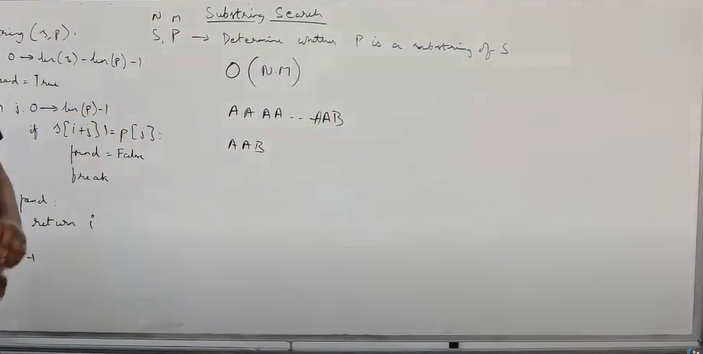
# Substring Search



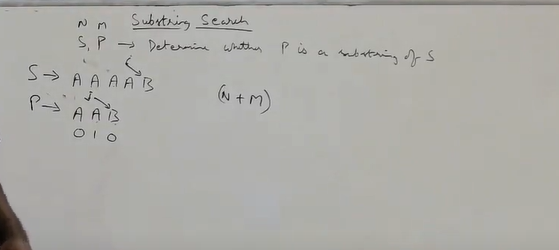
# Naïve Approach pattern searching



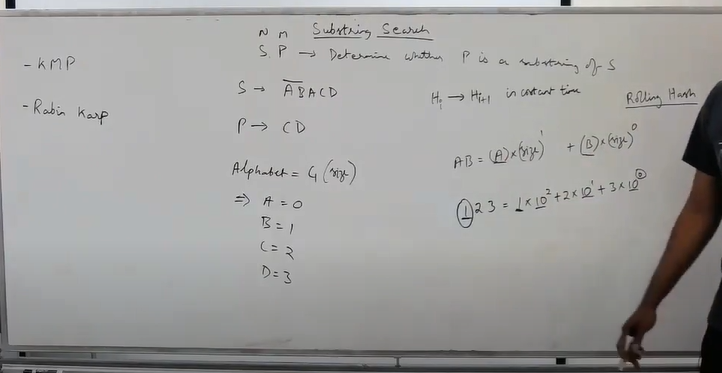
# Time Complexity of naïve approach – O(mn)



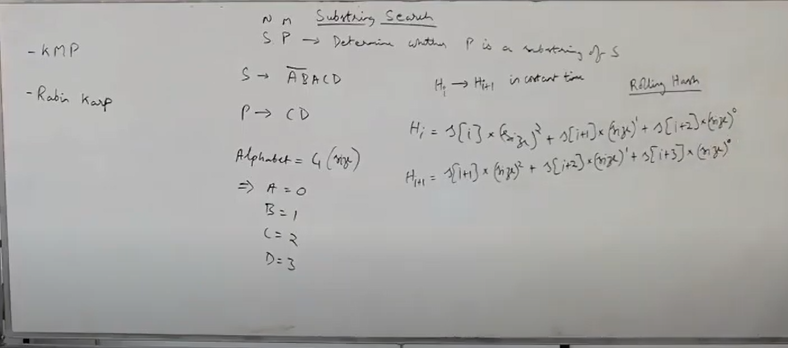
# KMP Time Complexity: O(n + m)

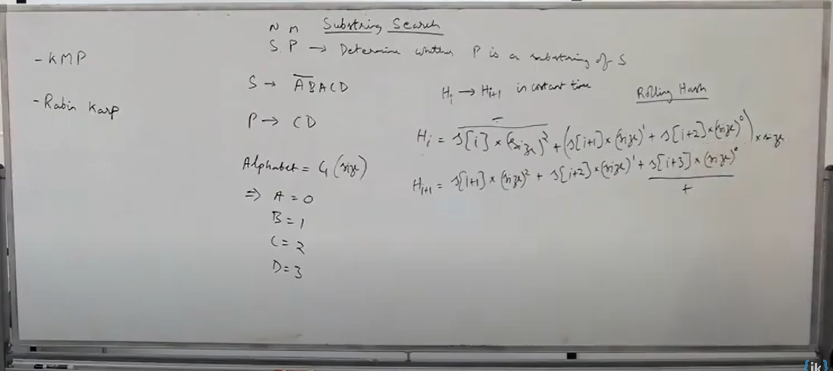


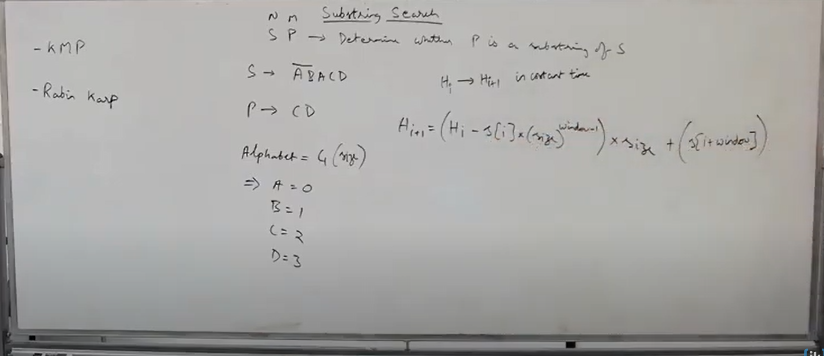
# Rabin karp



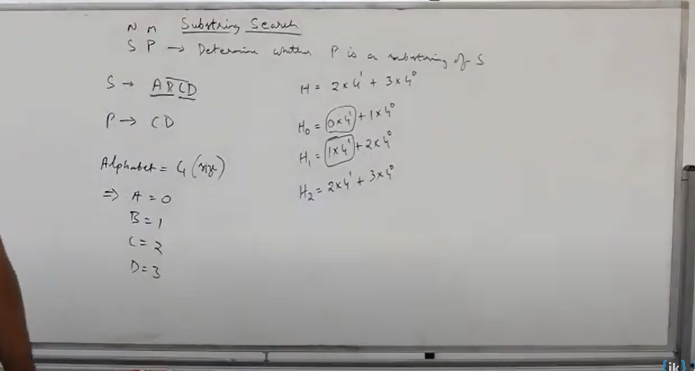
Calculate has for the first m characters.







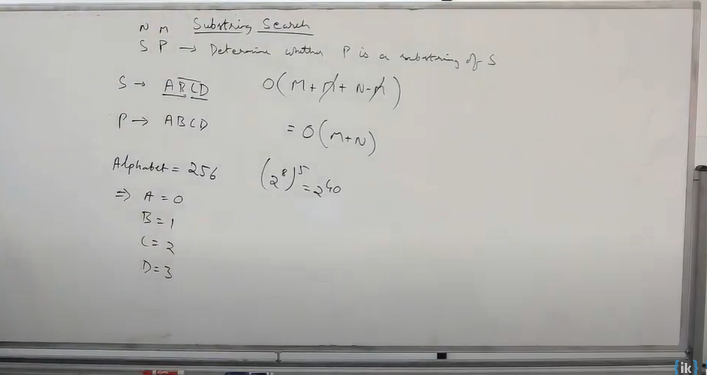
# Example Calculation

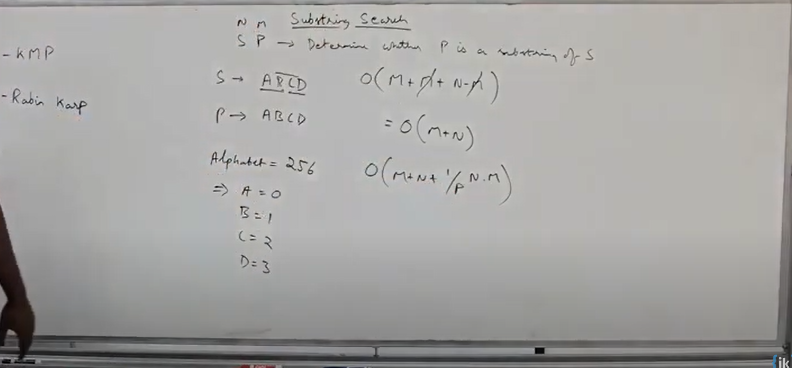


# Time Complexity:

O(m + n)

# Integer overflow





To avoid overflow, modulo with P where P = large prime number.

So, the hash value is contained between 0 to p – 1.

But then there is a possibility of collision and if collision happens, we end up doing comparison.

Let say that probability of collision is 1/P

Now this probability is over N

1/P \* N \*M because we compare M character to resolve the collision.

If we choose large enough P then we won’t have problem here, it will be negligible.

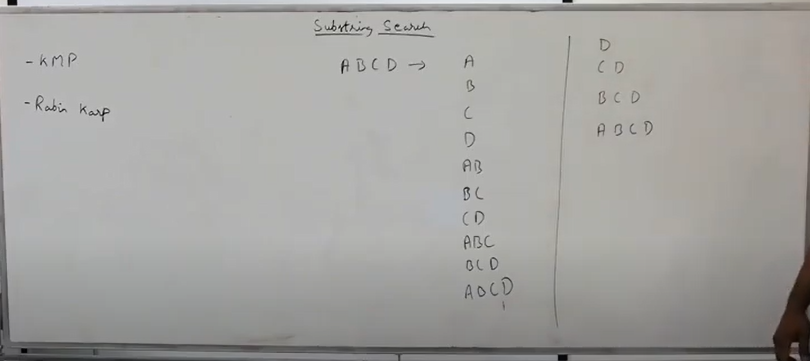
O(N + M + 1/P \* N \*M)

Applications:

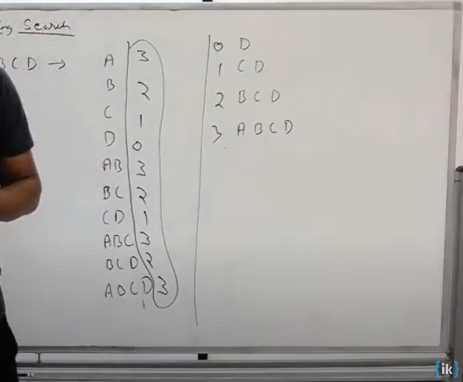
Stream of integers and identify if the pattern exists, you can use rolling hash.

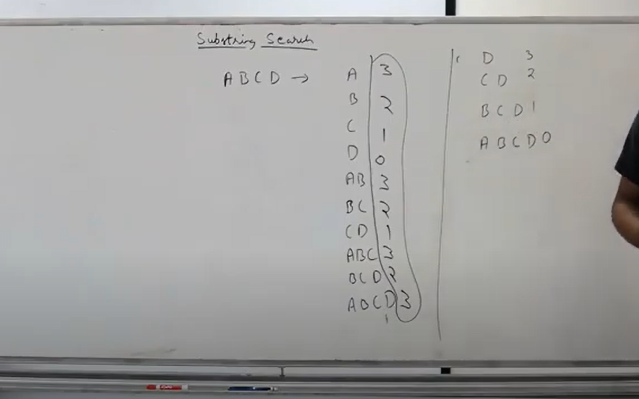
# Another approach to identify string is a substring of another

# Substrings and Suffix of String “ABCD”



# All of substrings are prefix of suffix

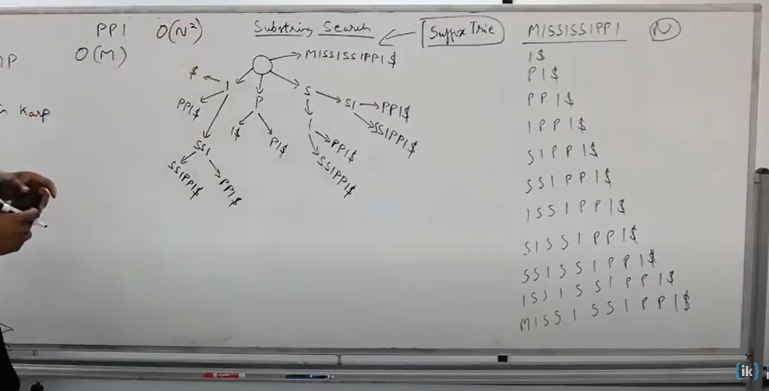




Why because every substring starts at one of this suffix index and end at or before D. Therefore, every substring starts at one of the suffix index and all substrings ends before or at D.

All of the suffix start at one of these indexes and end at D(end of the string)

# Suffix Tree - Construct Trie of all the suffixes then do the substring search inside it.

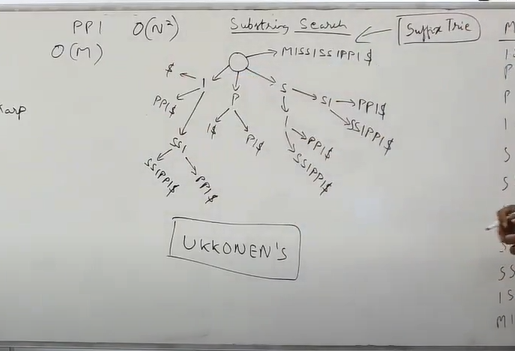


Construct Radix tree representation of trie.

The construction time is O(n^2) there are n suffixes and max length suffix is of n characters.

But the search is O(m)

If we have a string and many substring search queries coming in, then we can search in O(m) time.



You can build suffix tree in O(n) time using Ukkonen’s algorithm.

So overall time complexity is O(m + n)

To implement Ukkonen is very time consuming.

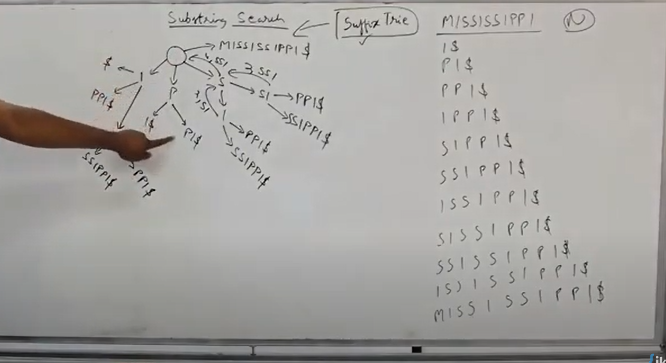
Build suffix tree using Trie in an interview setting and explain that you can also do this in linear time using Ukkonen.

Going through all the pain to build suffix tree, just to see if the substring exists doesn’t make sense.

We already have KMP and Rabin Karp algorithm which does that in linear time.

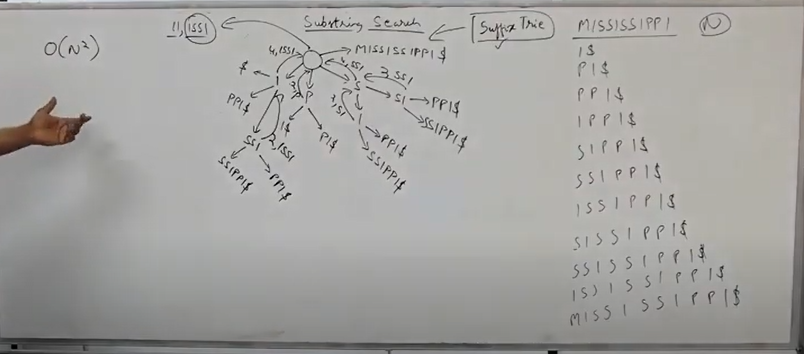
There are other use of suffix tree.

To find longest repeating substring



I am repeating if I have more than one dollar.

In “MISSISSIPPI” I is repeating 4 times, “SSI” is repeating 2 times



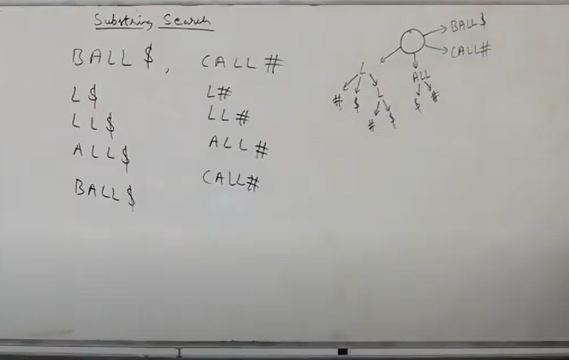
“ISSI” is the longest repeating substring. One tree traversal take O(n^2).

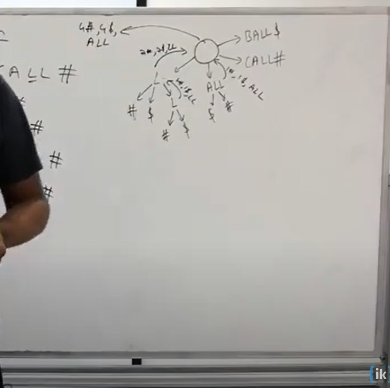
You can use trie to implement suffix tree, you don’t have to use radix tree.

Most repeating substring is “I” and “S” both repeats 4 times.

This suffix tree is for one word, what if we have more than one word.

For that we use a generalized suffix tree.





Which has both dollar and pound and longest substring.