

# Trie

15 September 2023

11:36 AM

## Implement TRIE | INSERT | SEARCH | STARTSWITH

Trie is used when we need insert[word], search[word], startwith[word] means any words whose prefix is word

Structure of Trie

Trie is a struct or class which has an array of 26 size of type className, bool flag  
In inside array 0 means a, 1, means b, 2 means c....., 25 means z

```
trie {  
    int a[26];  
    bool fl;  
}
```

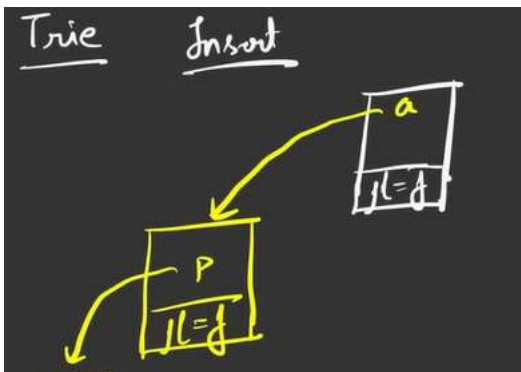
### Insert in Trie

Let say we take word apple

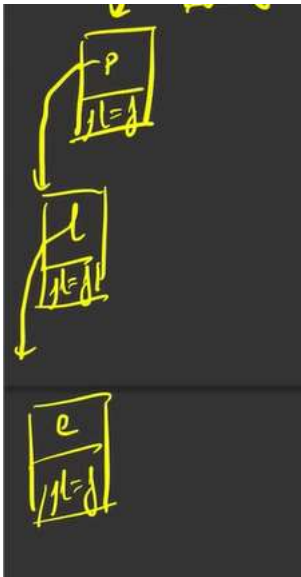
We see a letter if its not present we insert it in trie in a way that we create a trie node of arr[26] and bool flag and insert arr[1] = a and we create a reference trie of 'a'

Now we see next letter as 'p'

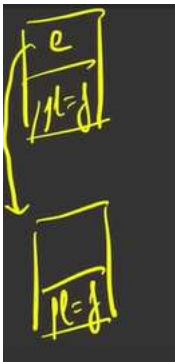
We check 'p' in trie and store It in reference of 'a' Trie node and create a reference trie node of 'p'



Sameway we do this for all left over characters



We insert 'e' at reference node of 'l' and we make a reference trie of 'e' also  
Initially flag = false

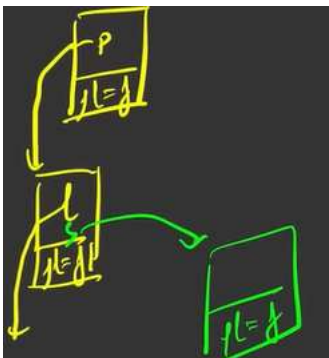


We are at 'e' means word is completed so we mark flag = true for reference trie node of 'e'  
We have finished for 'apple'

Now let's see for 'apps'

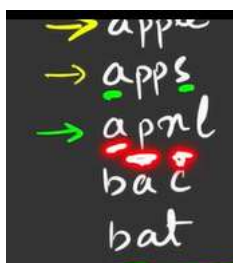
We check in 'apple' and we find that 'app' are found but no 's' found

So we insert 's' in array of trie node below 'p' which is 'l' and make a ref trie node in reference node of 'p' which is after 'l' node in trie with bool false



apps is complete so we mark flag = true for new reference node created below 'l'  
Which shows that this is another word

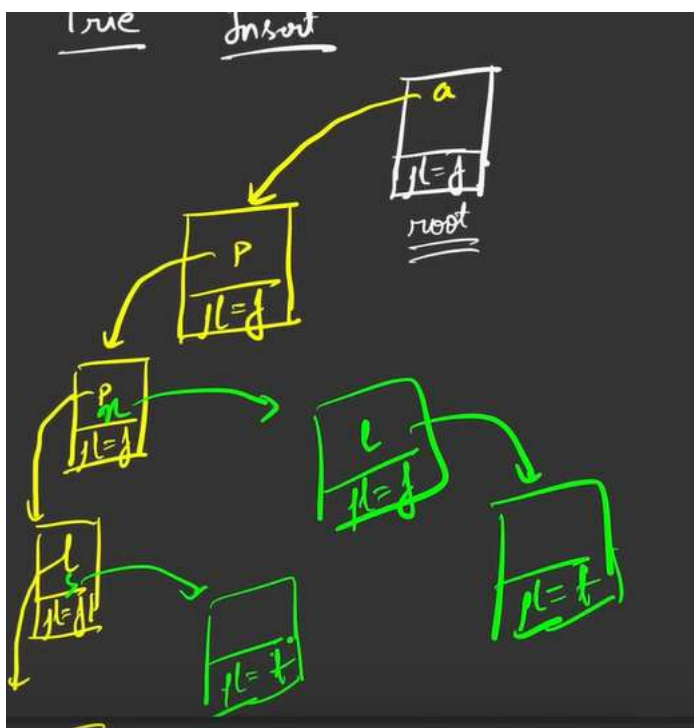
Now we check for



"apxl" we see we don't have x so we create x in array of reference node of 'p' in apple

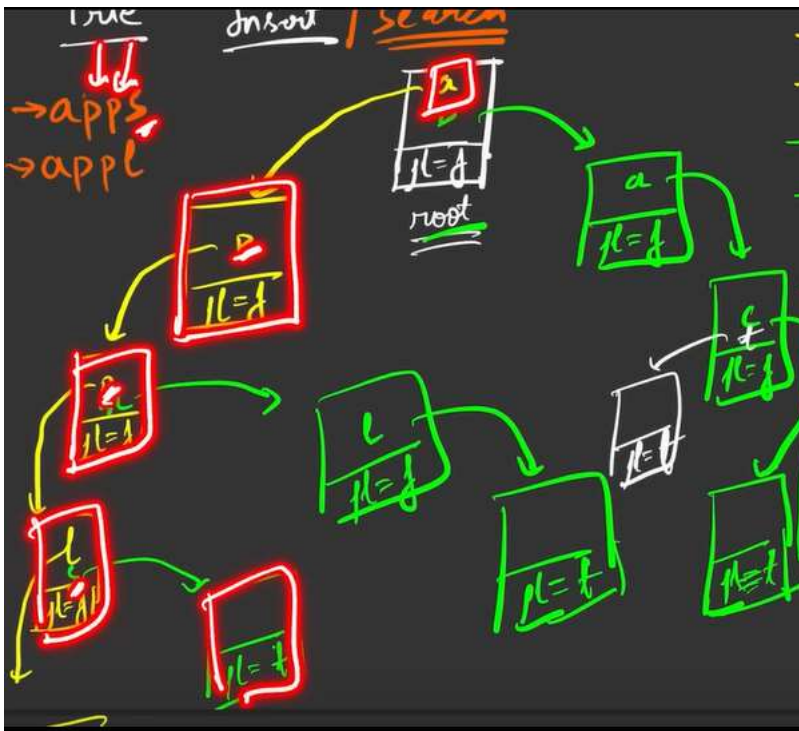
And we need 'l' now after x so we create one more node

Word is completed so mark `bool = true` which shows one more word done.



Now we go for "bac" then "bat"



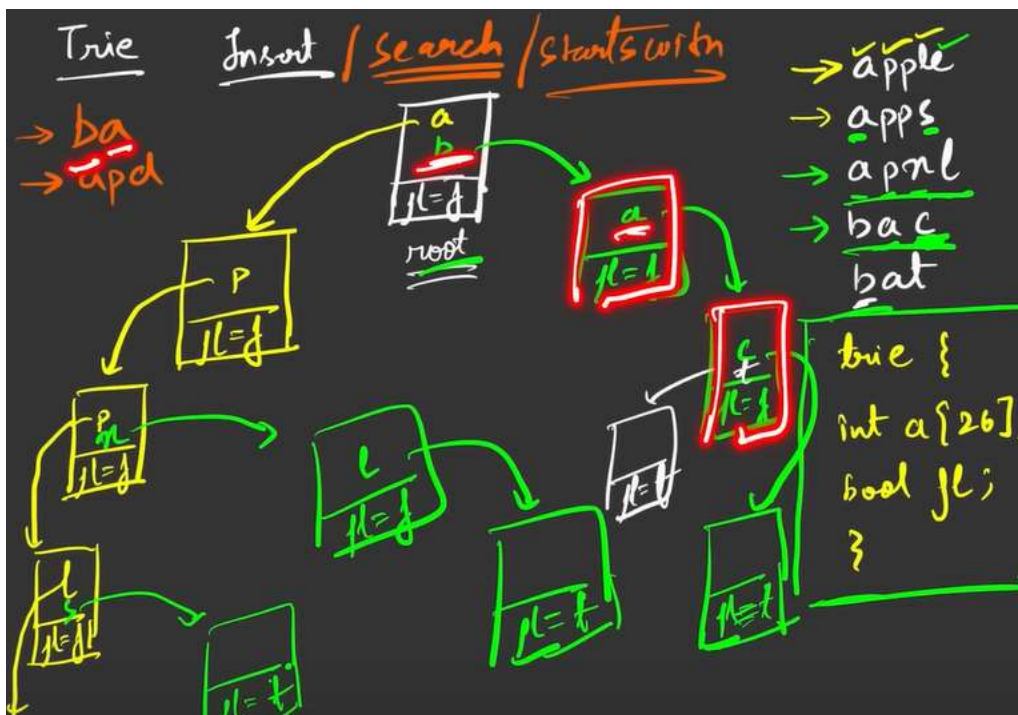


## StartsWith

Is there any word which starts with "ba"

No, we start from root and we see there is "b" and we have "a" we move ahead

If we are standing at a trie node and its **not NULL** means there is such word with prefix "ba"



If we are encountering a NULL means there does not exists any such word

## Code

```

struct Node{
    Node *arr[26];
    bool flag = false;
    // To check if our array already has ch or not before inserting
    bool hasLetter(char ch)
    {
        // if node of that character is not NULL means that exists
        return (arr[ch - 'a'] != NULL);
    }
    // To insert a node in array for ch
    void put(char ch, Node *node)
    {
        arr[ch-'a'] = node;
    }
    // To get ref node of ch
    Node* get(char ch)
    {
        return arr[ch-'a'];
    }
    // To set the end ref node ka bool as true
    void setEnd()
    {
        flag = true;
    }
    // Check if end ref node has flag value = true means word complete
    // else false, then word is not found completely
    bool isEnd()
    {
        return flag;
    }
};

class Trie {
private:
    Node* root;
public:
    /** Initialize your data structure here. */
    Trie() {
        // Every Time constructor is called, make a new root
        root = new Node();
    }

    /** Inserts a word into the trie. */
    void insert(string word) {
        // TC: O(length of word)
        // we need to insert each letter of word in trie
        // starting from root
        Node* node = root;
        for(int i = 0; i < word.size(); i++)
        {
            // before inserting check if that already exist so we make
            // a function in struct Node
            if(!node->hasLetter(word[i]))
            {
                // we have used ! means if we here, it does not exist
                // let us insert it
                node->put(word[i], new Node());
            }
        }
    }
};

```

```

    }
    // move to reference of trie node
    node = node->get(word[i]);
}
// loop ends means we are at the last ref node means mark the
bool = true for last ref node as word is finished
node->setEnd();
}

/** Returns if the word is in the trie. */
bool search(string word) {
    // TC: O(length of word)
    // always start with root node
    Node* node = root;
    for(int i = 0; i < word.size(); i++)
    {
        // check if char is there or not
        if(!node->hasLetter(word[i]))
        {
            // if it does not exists means
            return false;
        }
        // else give me ref node of that char
        // we are storing ref node so that we can check at the end
        if last ref node is true means complete word is found
        node = node->get(word[i]);
    }
    // now we are at end ref node so check bool
    // if true means we are at the end else not
    if(node->isEnd())
    {
        return true;
    }
    return false;
}

/
/** Returns if there is any word in the trie that starts with the given
prefix. */
bool startsWith(string word) {
    // TC: O(length of word)
    // again start from root
    Node* node = root;
    for(int i = 0; i < word.size(); i++)
    {
        if(!node->hasLetter(word[i]))
        {
            return false;
        }
        node = node->get(word[i]);
    }
    // if we reach here means there is something whose prefix is w
ord
    return true;
}
};

```

## Implement Trie-2 | INSERT | countWordsEqualTo() | countWordsStartingWith()

Ninja has to implement a data structure "TRIE" from scratch. Ninja has to complete some functions.

1) Trie(): Ninja has to initialize the object of this "TRIE" data structure.

2) insert("WORD"): Ninja has to insert the string "WORD" into this "TRIE" data structure.

3) countWordsEqualTo("WORD"): Ninja has to return how many times this "WORD" is present in this "TRIE".

4) countWordsStartingWith("PREFIX"): Ninjas have to return how many words are there in this "TRIE" that have the string "PREFIX" as a prefix.

5) erase("WORD"): Ninja has to delete one occurrence of the string "WORD" from the "TRIE".

Note:

1. If erase("WORD") function is called then it is guaranteed that the "WORD" is present in the "TRIE".

2. If you are going to use variables with dynamic memory allocation then you need to release the memory associated with them at the end of your solution.

**Sample Input 1:**

```
1
5
insert coding
insert ninja
countWordsEqualTo coding
countWordsStartingWith nin
erase coding
```

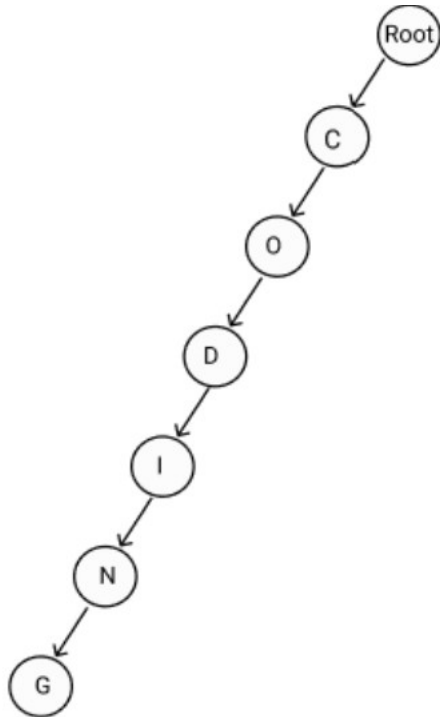
**Sample Output 1:**

```
1
1
```

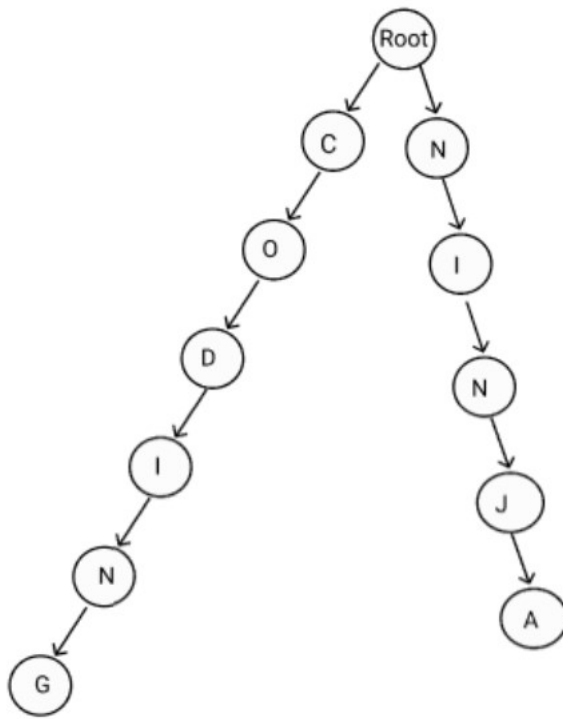


### Explanation Of Sample Input 1:

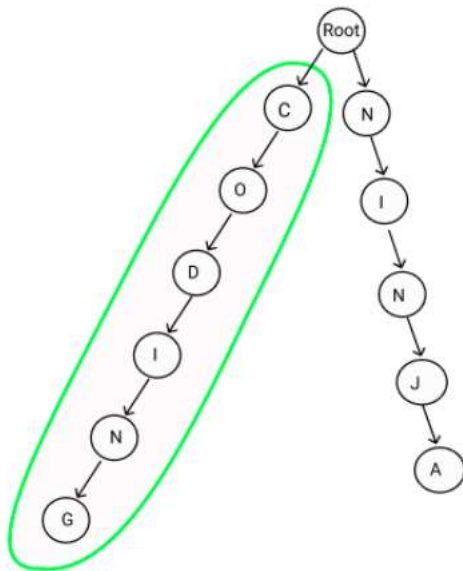
After insertion of "coding" in  
"TRIE":



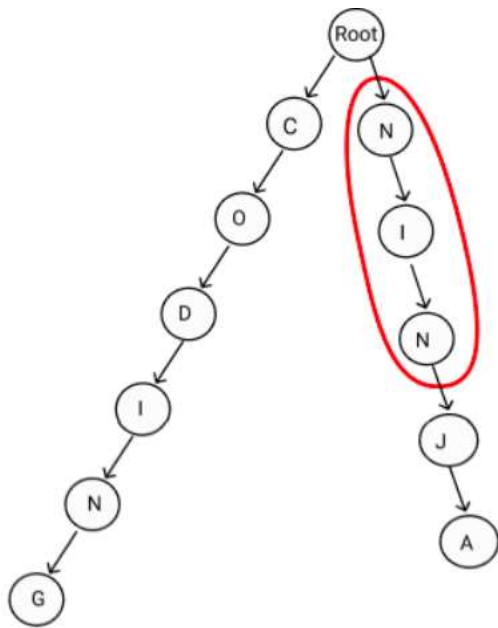
After insertion of "ninja" in  
"TRIE":



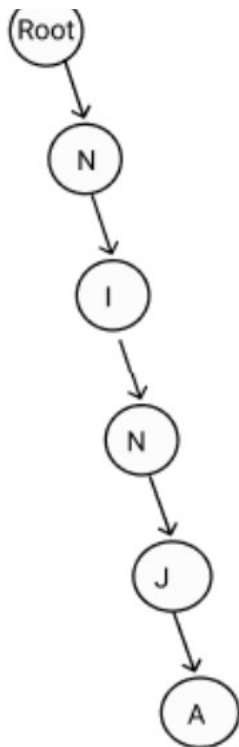
Count words equal to "coding" :



Count words those prefix is "nin":



After deletion of the word "coding",  
"TRIE" is:



**Sample Input 2:**

1

6

insert samsung

insert samsung

insert vivo

erase vivo

countWordsEqualTo samsung

countWordsStartingWith vi

**Sample Output 2:**

2

0

**Explanation for sample input 2:**

insert "samsung": we are going to insert the word "samsung" into the "TRIE".

insert "samsung": we are going to insert another "samsung" word into the "TRIE".

insert "vivo": we are going to insert the word "vivo" into the "TRIE".

erase "vivo": we are going to delete the word "vivo" from the "TRIE".

countWordsEqualTo "samsung": There are two instances of "samsung" is present in "TRIE".

countWordsStartingWith "vi": There is not a single word in the "TRIE" that starts from the prefix "vi".

## Approach

This time our requirement is different so we need to modify our trie accordingly.

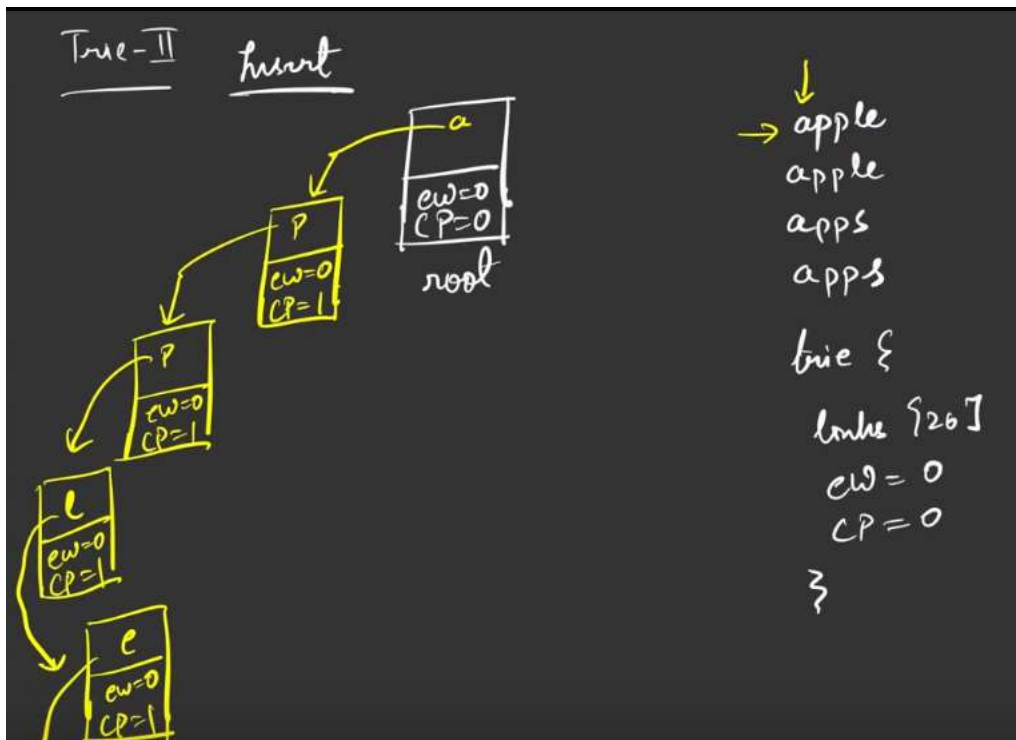
We will again have array of 26 but instead of bool flag, we will have a variable endWith = 0, Variable countPrefix = 0.

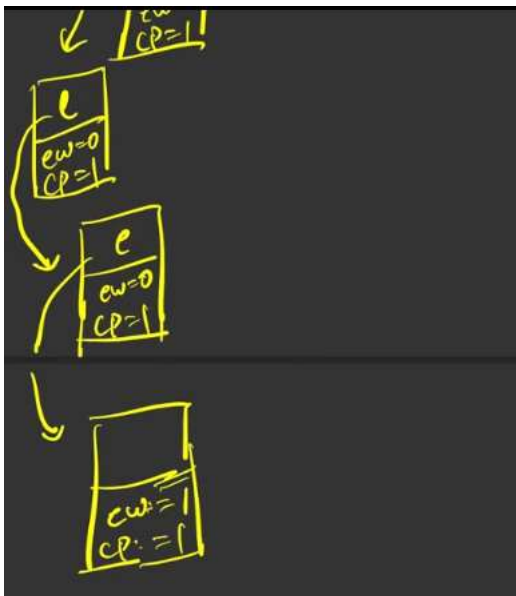
We will initially declare a root with countPrefix (CP) = 0, endWith (EW) = 0

## Insert

We will insert "apple"

At every node, ew = 0, cp = +1





At the end node, ew = 1,

Now go for next test case "apple" in apple

We see "a" yes it is there, so go to ref node of "a" i.e "p" and mark it CP = 1 se 2

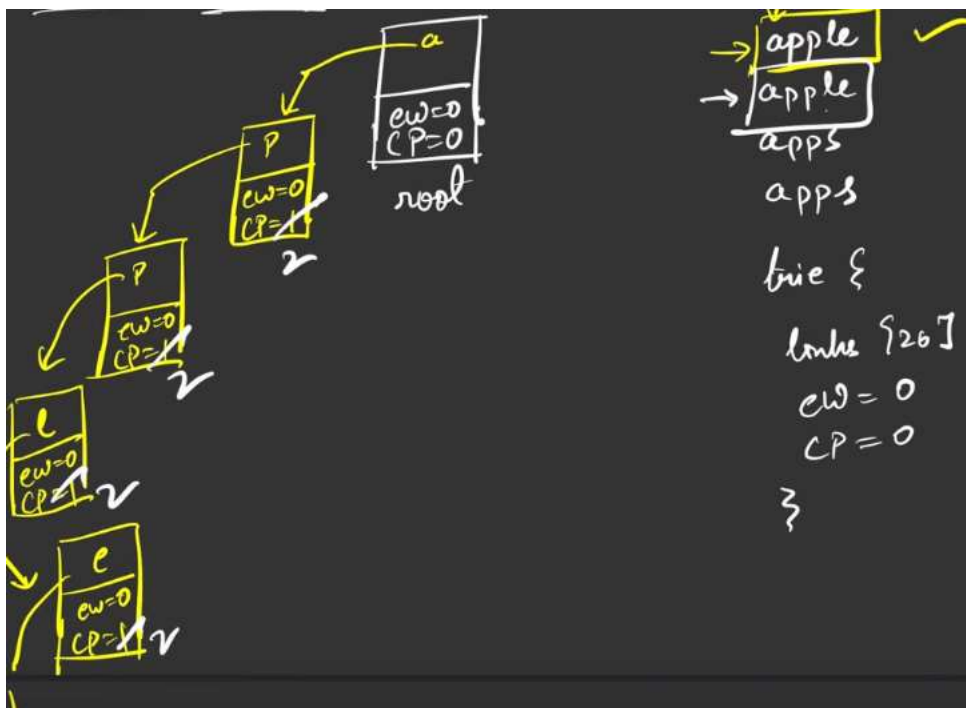
Now we see "p" yes we found it, go to its ref node and make CP = 1->2

Now again we see "p" yes we found it, go to its ref node and make CP = 1->2

Now we see "l", we found do the same

Now we see "e", we found do the same with last node, ew = 1, cp = 2

Now we are at last node



We check for "apps", we increase CP = 3 for "a", "p", "p" but we did not found "s" so insert "s" in ref node of "p" with CP = 1, EW = 0

Now "apps" ends so make sure to make EW = 1 for "apps" last node



## Count Words Start with( )

Check character by character and last character ka ref node ka EW will tell, number of word starts with( )

## Erase( )

Start from root, and as you found character by character, reduce the CP of ref node of each character.

When you reach end of word, reduce EW of last node also

## Code

```
struct Node{
    Node* arr[26]; // 0-based indexing
    int endWith = 0;
    int countPrefix = 0;

    bool containsLetter(char ch)
    {
        return (arr[ch - 'a']);
    }

    void put(char ch, Node* node)
    {
        arr[ch-'a'] = node;
    }

    Node* get(char ch)
    {
        return arr[ch - 'a'];
    }

    void incrementEndWith()
    {
        endWith++;
    }

    void decrementEndWith()
    {
        endWith--;
    }

    void incrementCountPrefix()
    {
        countPrefix++;
    }

    void decrementCountPrefix()
    {
        countPrefix--;
    }

    int getEndWithofLastNode()
```

```

    {
        return endWith;
    }

    int getCountPrefix()
    {
        return countPrefix;
    }
};

```

```

Class Trie{

```

```

    private:

```

```

    Node* root;

```

```

    public:

```

```

    Trie(){
        root = new Node;
    }

```

```

    void insert(string &word){
        Node* node = root;
        for(int i = 0; i < word.size(); i++)
        {
            if(!node->containsLetter(word[i]))
            {
                node->put(word[i], new Node());
            }
            node = node->get(word[i]);
            node->incrementCountPrefix();
        }
        // last node
        node->incrementEndWith();
    }

```

```

    int countWordsEqualTo(string &word){
        // start from root
        Node* node = root;
        for(int i = 0; i < word.size(); i++)
        {
            if(node->containsLetter(word[i]))
            {
                node = node->get(word[i]);
            }
            else{
                return 0;
            }
        }
        // End with has info of number of words ending with this word
        return node->getEndWithofLastNode();
    }

```

```

    int countWordsStartingWith(string &word){
        // start from root
        Node* node = root;

```



```

        for(int i = 0; i < word.size(); i++)
        {
            if(node->containsLetter(word[i]))
            {
                node = node->get(word[i]);
            }
            else{
                return 0;
            }
        }
        // Count Prefix has information of number of word whose prefix
        is this word
        return node->getCountPrefix();
    }

    void erase(string &word){
        // start from root
        Node* node = root;
        for(int i = 0; i < word.size(); i++)
        {
            // Assume we found it, so decrease CP for everyone
            if(node->containsLetter(word[i]))
            {
                node = node->get(word[i]);
                node->decrementCountPrefix();
            }
            else{
                return;
            }
        }
        // we are at last node
        // so decrease EW also
        node->decrementEndWith();
    }
};

```

## Complete String

Ninja developed a love for arrays and strings so this time his teacher gave him an array of strings, 'A' of size 'N'. Each element of this array is a string. The teacher taught Ninja about prefixes in the past, so he wants to test his knowledge.

A string is called a complete string if every prefix of this string is also present in the array 'A'. Ninja is challenged to find the longest complete string in the array 'A'. If there are multiple strings with the same length, return the lexicographically smallest one and if no string exists, return "None".

Note:

String 'P' is lexicographically smaller than string 'Q',

if :

1. There exists some index 'i' such that for all 'j' < 'i', 'P[j] = Q[j]' and 'P[i] < Q[i]'. E.g. "ninja" < "noder".
2. If 'P' is a prefix of string 'Q', e.g. "code" < "coder".

Example:

N = 4

A = [ "ab" , "abc" , "a" , "bp" ]

Explanation :

Only prefix of the string "a" is "a" which is present in array 'A'. So, it is one of the possible strings.

Prefixes of the string "ab" are "a" and "ab" both of which are present in array 'A'. So, it is one of the possible strings.

Prefixes of the string "bp" are "b" and "bp". "b" is not present in array 'A'. So, it cannot be a valid string.

Prefixes of the string "abc" are "a", "ab" and "abc" all of which are present in array 'A'. So, it is one of the possible strings.

We need to find the maximum length string, so "abc" is the required string.

### Approach

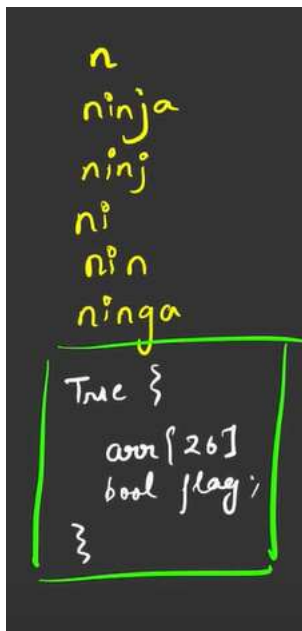
If every prefix of string is also present in array A

If there are multiple strings, return lexicographically smaller one

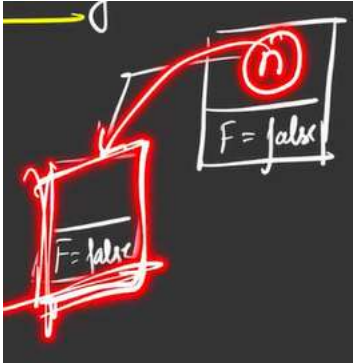
If there does not exist then return none

I will insert all elements of array in Trie

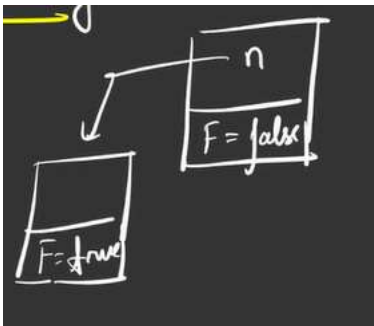
Now I will check element by element if that word exists in trie, I store it



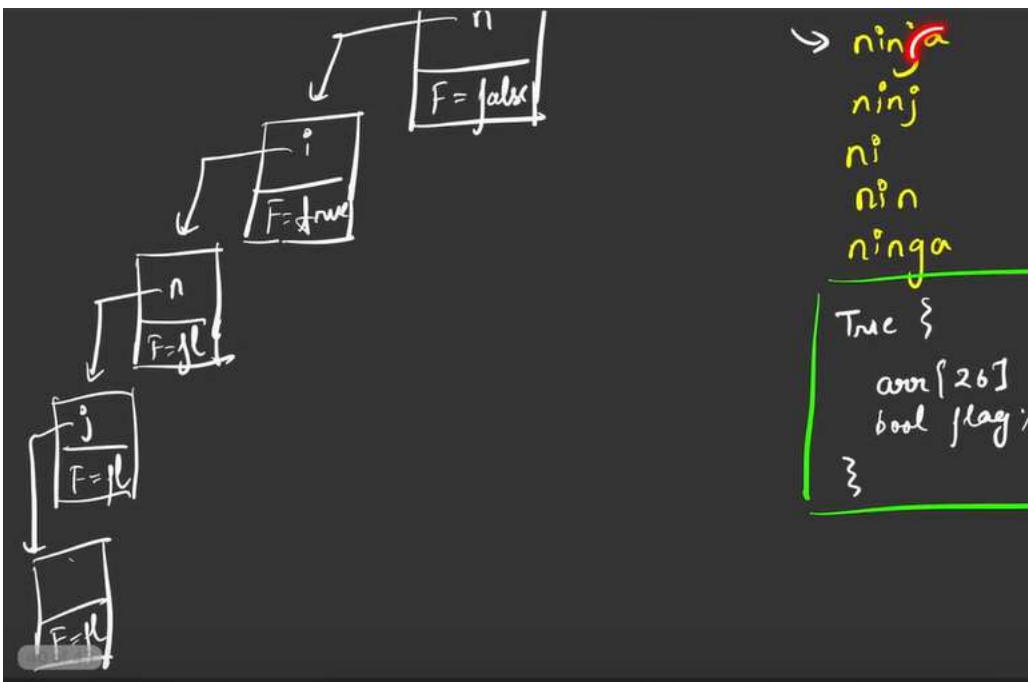
We take root of the trie and take flag = false initially, first word is "n"  
Its not there so insert It and a ref node is created



Now am standing at ref node of "n" and first element of array is done so I mark ref node ka flag = true

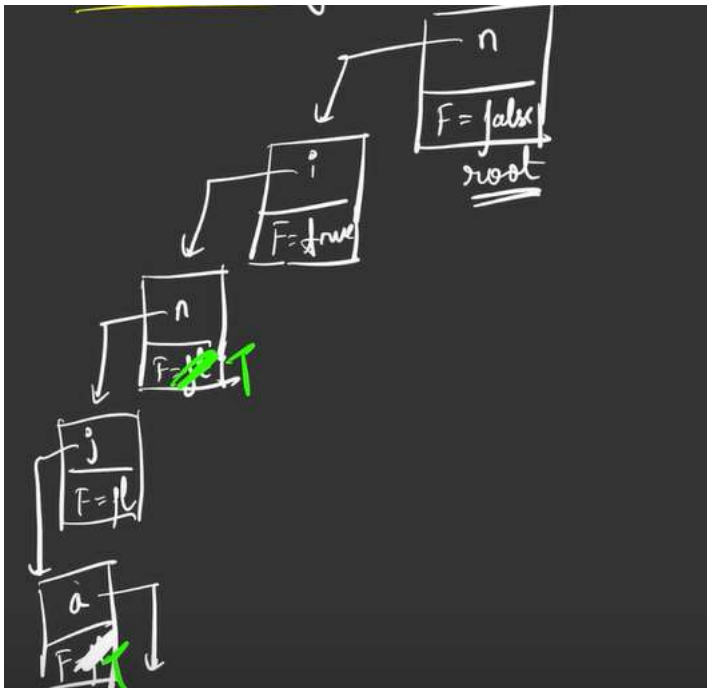


Let's go to "ninja" next element  
I have "n" but I don't have "i" insert it  
Sameway insert "n", "j", "a" also now we are at last node, we mark it true



Now we go to "ninj"

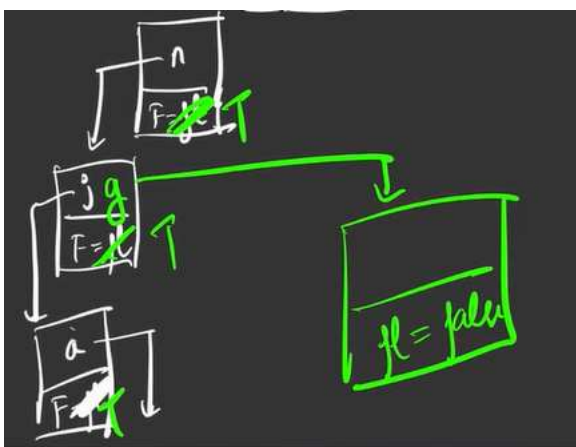
Start from root and we found "n", next go to ref node of it, we found "l" also  
 Now next found "n" also, found "j" also but ref node of "j" flag = false so mark it true and  
 we can say "ninj" also insert



Do same for all elements and mark true when one element ends



If we do not have any node, we insert it and make a ref node like this  
 In "ninga" we don't have "g" so insert "g" in ref node of "n"



Now insert "a" in ref node of "g" and make ref node of "a" with flag = false  
So words end means flag = true

**All words are inserted Now..**

Let's again traverse array and check element by element, make a variable complete string (CS)

For "n", yes it's a complete string so CS = n

For "ninja", check "n" yes it exists and its ref node is true means there exists a word which is "n"

Check "i", yes exists and ref node is true means there is something "ni" in array

Check "n", yes there exist something as "nin" in array

Same for "j", "a" and we see CS = ninja now and this can be a possible answer

Do same for "ni", "nin", "ninga"

We see longest common string is "ninja"

**Complexity:**

TC:  $O(n) * O(\text{length of longest string})$  to insert +  $O(N * \text{length})$  for checking

SC: its difficult to predict because every try use array of 26 element

**Code:**

```
// Let us make a struct node
struct Node{
    Node* arr[26];
    bool flag = false;
    bool containsKey(char ch)
    {
        return (arr[ch-'a'] != NULL);
    }
    Node* get(char ch)
    {
        return arr[ch - 'a'];
    }
    void put(char ch, Node* node)
    {
        arr[ch-'a'] = node;
    }

    void setEnd()
    {
        flag = true;
    }
    bool getEnd()
    {
        return flag;
    }
};
```

```
class Trie{
```

```

private:
Node* root;

public:
// constructor
Trie()
{
    root = new Node();
}

void insert(string word)
{
    // start from root
    Node* node = root;
    for(int i = 0; i < word.size(); i++)
    {
        // check kar
        if(!node->containsKey(word[i]))
        {
            node->put(word[i], new Node());
        }
        node = node->get(word[i]);
    }
    // At last mark last node as true flag
    node->setEnd();
}

bool wordisThere(string word)
{
    // starting from root
    Node* node = root;
    bool flag = true;
    for(int i = 0; i < word.size(); i++)
    {
        if(node->containsKey(word[i]))
        {
            node = node->get(word[i]);
            flag = flag && node->getEnd();
        }
        else{
            return false;
        }
    }
    return flag;
}

};

string completeString(int n, vector<string> &a){
    // Make object of Trie class
    Trie obj;
    string complete = "";
    for(auto it: a)
    {
        // Insert all strings in trie
        obj.insert(it);
    }
}

```

```

    }
    // Again traverse and look for every word
    for(auto it: a)
    {
        if(obj.wordisThere(it))
        {
            // compare with complete because we need longest string
            if(it.size() > complete.size())
            {
                complete = it;
            }
            else if(it.size() == complete.size() && it < complete)
            {
                // if length of both are same means return lexo smaller
                complete = it;
            }
        }
    }
    if(complete == "") return "None";
    return complete;
}

```

## Count Distinct Substrings

Given a string 'S', you are supposed to return the number of distinct substrings(including empty substring) of the given string. You should implement the program using a trie.

Note:

A string 'B' is a substring of a string 'A' if 'B' that can be obtained by deletion of, several characters(possibly none) from the start of 'A' and several characters(possibly none) from the end of 'A'.

Two strings 'X' and 'Y' are considered different if there is at least one index 'i' such that the character of 'X' at index 'i' is different from the character of 'Y' at index 'i' ( $X[i] \neq Y[i]$ ).

**Sample Input 1 :**

2  
sds  
abc

**Sample Output 1 :**

6  
7

**Explanation of Sample Input 1 :**

In the first test case, the 6 distinct substrings are { 's', 'd', "sd", "ds", "sds", "" }

In the second test case, the 7 distinct substrings are { 'a', 'b', 'c', "ab", "bc", "abc", "" }.

**Sample Input 2 :**

2  
aa  
abab

**Sample Output 2 :**

3  
8

### Explanation of Sample Input 2 :

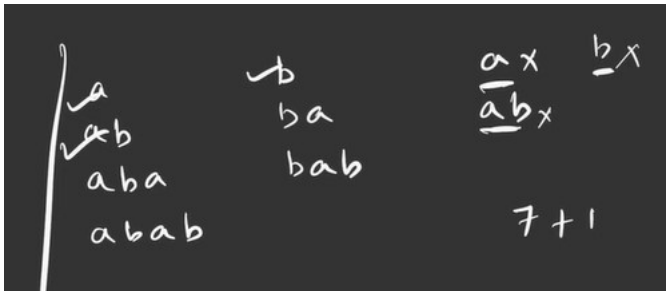
In the first test case, the two distinct substrings are {'a', "aa", "" }.

In the second test case, the seven distinct substrings are {'a', 'b', "ab", "ba", "aba", "bab", "abab", "" }

### Approach

Let say word is "abab"

Let us write all substring from "a", "b", "a", "b"

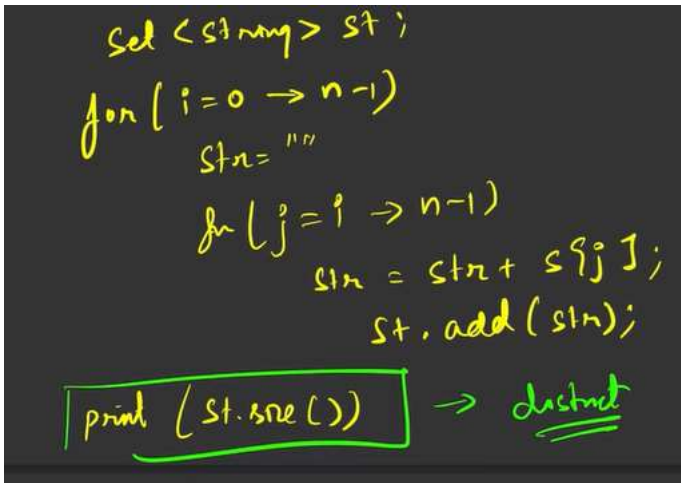


But we need distinct substrings so we neglect a,ab,b which are repeating

**According to problem statement, We have to take one empty string also so answer is 7 + 1 = 8**

## Brute force

Pick a character and print all substrings, use a set to get distinct/ unique substrings and return set.size at end to give number of such distinct substrings



TC:  $O(n^2 * \log nM)$

SC: Let say all substrings generated are distinct

Min length can be 1, max length will be n for any substring say  $n/2$  in average case

Say we have  $n^2$  substrings all distinct inside set

Each substring takes  $O(n/2)$  space so  $O(n^3)$

We can use String Hashing algos like rabin karp etc

We can also use Trie, we only need 4 trie for a word of 4 size

**We will use Trie**



Let say our string is "abab"

We will see if string exists in Trie, if not, we insert it and we will check if it's a word or not

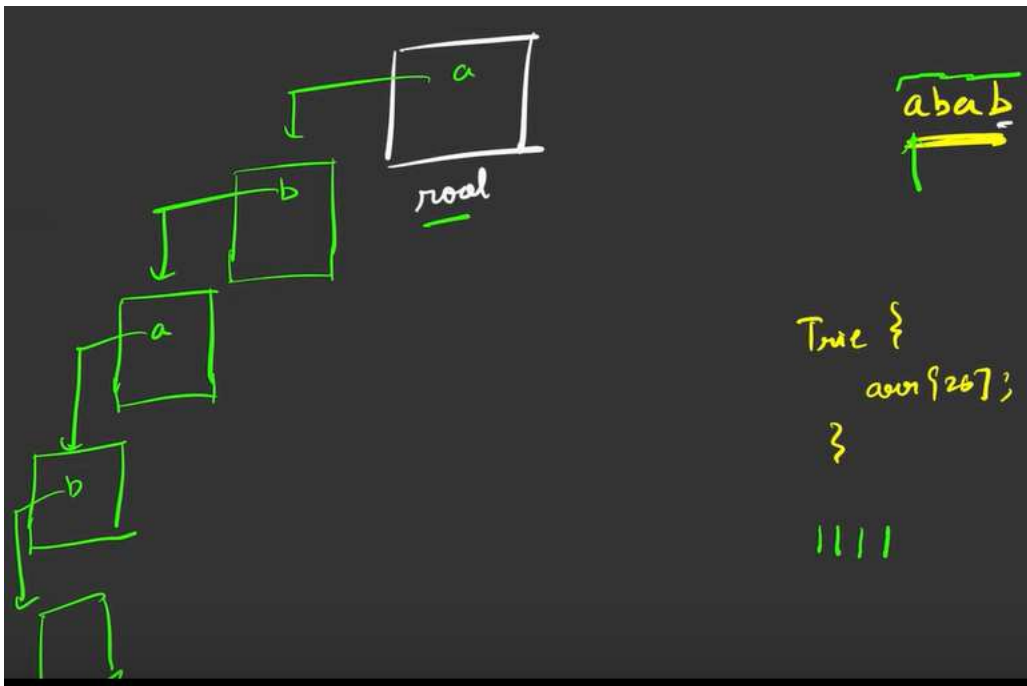
Start with "a" from "abab"

We keep a count, we see something not present, we insert it and make its ref node and count++

Go to "ab", a is there but b is not so insert "b" and do count++

For "aba", insert "a" again so count++

For "abab", insert "b" again so count++

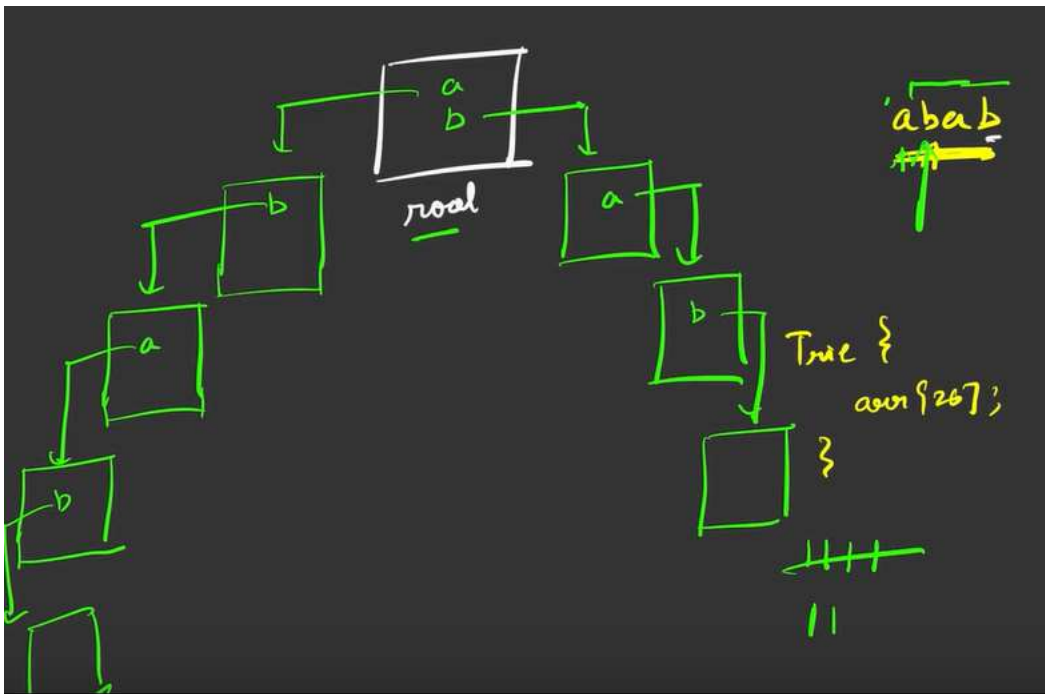


Now we start from "b" from "abab"

We see "b" not present at root, insert and count++

We see "ba" not there so insert count++

Same for "bab" and count++

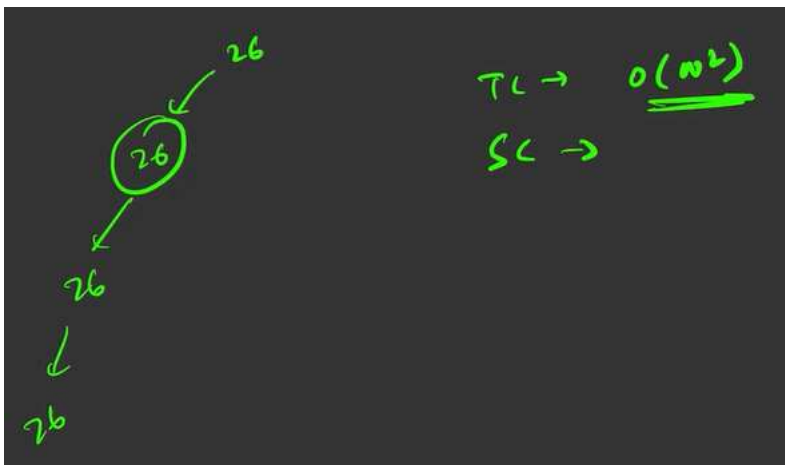


Now start for "a" of "ab**a**b"

If it already exists not need to do count++. Same we do for "b" and at the end we do +1 for empty string

TC:  $O(N^2)$  for inserting

SC: its difficult for Trie, 26 -> 26 -> 26.....



## Code:

```
struct Node{
    Node* arr[26];
    void put(char ch, Node* node)
    {
        arr[ch - 'a'] = node;
    }
    Node* get(char ch)
    {
        return arr[ch - 'a'];
    }
}
```

```
int countDistinctSubstrings(string &s)
{
    // take a counter
    // we are not making Trie class here because we only need insert f
unction which can be made here also
    int count = 0;
    // start from root
    Node* root = new Node;
    for(int i = 0; i < s.size(); i++)
    {
        Node* node = root;
        for(int j = i; j < s.size(); j++)
        {
            if(!node->containsKey(s[j]))
            {
                // if its does not contains, insert and increase count
                count++;
                node->put(s[j], new Node());
            }
            // get me to the ref node now
            node = node->get(s[j]);
        }
    }
    // +1 is for empty string
    return count + 1;
}
```

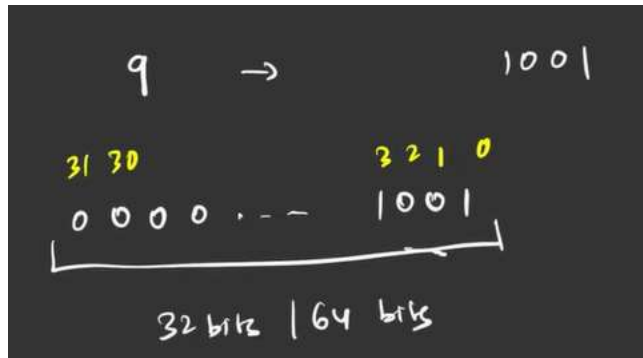
## Bit PreRequisites for TRIE Problems

In binary rep of 9 we write it as 1001

But we does not store it in Trie, this way

In trie we store the way, computer stores is i.e 000000000000000000000000**1001** for int (32 bits) and 00000000000000..000000000000**1001** for long long (64 bits)

And indexing starts from rightmost (0-based) to left



## What is XOR?

XOR of different number is 1

$$1 \wedge 0 = 1$$

$$0 \wedge 1 = 1$$

XOR of similar number is 0

$$1 \wedge 1 = 0$$

$$0 \wedge 0 = 0$$

Even number of 1's it gives = 0

Odd number of 1's it gives = 1

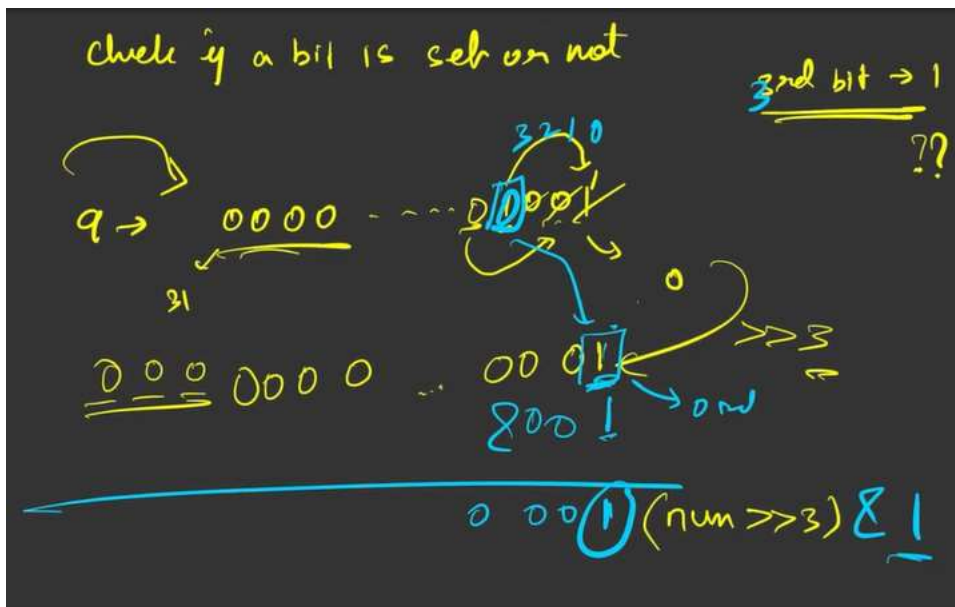
## Check bit is set (1) or not (0) ?

If we want to check 3rd bit of a number is set or not

We do right  $\gg 3$

What happens is 3rd bit will come in front now, all other will be 0

Now we do  $\& 1$  and if answer is 1 means bit is set



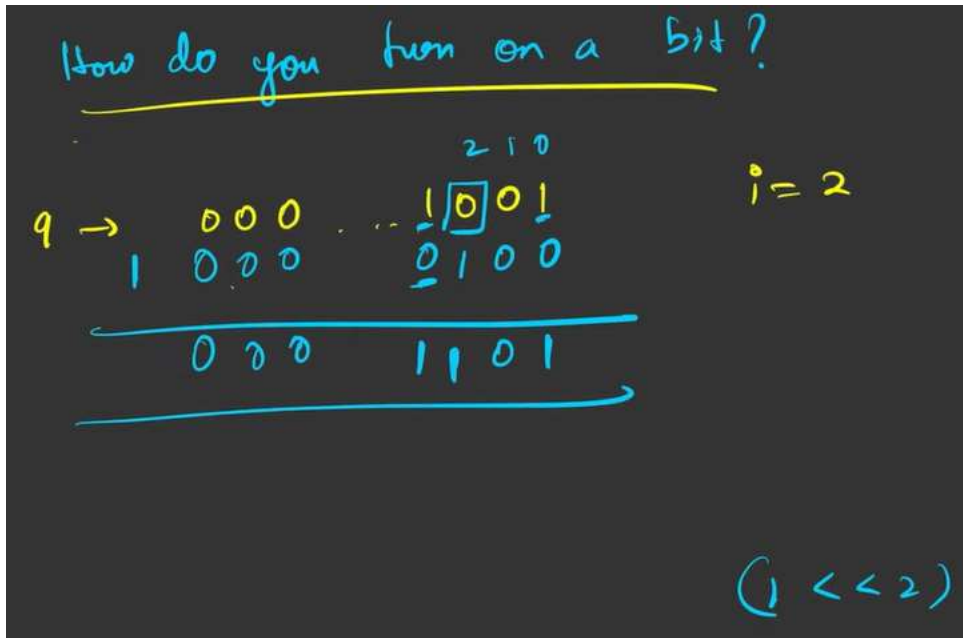
## A bit is set if

$$((num \gg i) \& 1) \neq 0$$

## And bit is not set if its = 0

## How do you turn on a bit?

We just do  $(1 \ll 2)$  and do OR with number



## Maximum XOR

You are given two arrays of non-negative integers say 'arr1' and 'arr2'. Your task is to find the maximum value of ( 'A' xor 'B' ) where 'A' and 'B' are any elements from 'arr1' and 'arr2' respectively and 'xor' represents the bitwise xor operation.

### Sample Input 1:

1  
7 7  
6 6 0 6 8 5 6  
1 7 1 7 8 0 2

### Sample Output 1:

15

### Explanation of sample input 1:

First testcase:

Possible pairs are (6, 7), (6, 8), (6, 2), (8, 7), (8, 8), (6, 2). And 8 xor 7 will give the maximum result i.e. 15

### Sample Input 2:

1  
3 3  
25 10 2  
8 5 3

### Sample Output 2:

28

### Explanation of sample input 2:

First test case:

28 is the maximum possible xor given by pair = (25, 5). It is the maximum possible xor among all possible pairs.

## Approach

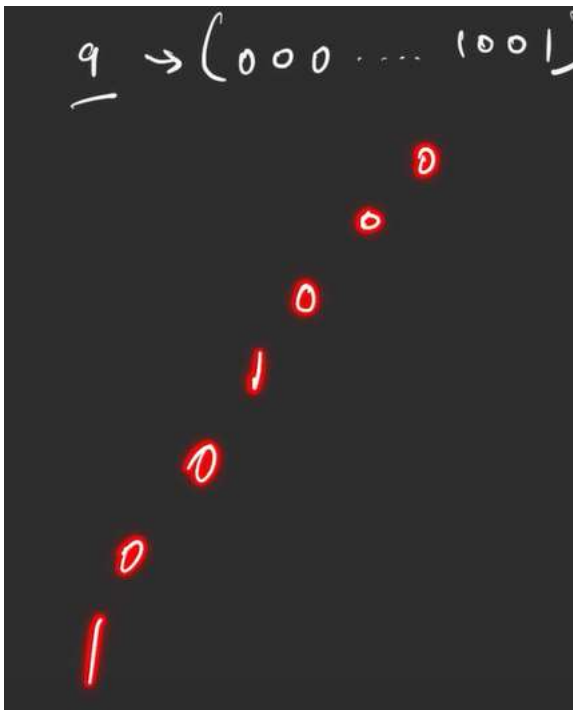
To understand/ solve above problem we need to understand below problem first

Q Given an array of numbers, & a number  $x$ .  
Find the max value of  $(arr[i] \wedge x)$ .

If we are given an array of number, we insert all number in Trie in terms of binary bits,  
We take  $x$  and find max number from array where  $number \wedge x$  is maximum

→ Insert all the numbers into the trie.  
    ↳ (binary bits)  
→ Take  $x$  & find the max no. from array  
    where  $(no \wedge x) \uparrow \uparrow$

We will not insert 9 directly, instead of 9 we insert 000000000000.....1001  
We will store it like a string

9 → (000 ..... 1001)  


We cannot draw 32 bits trie, let solve for 01001 and actually it will be 32 bits  
Our trie will have array of 2 node, 0 → 0 and 1 → 1

**Insert**

Let say array Is [9,8,7,5,4]

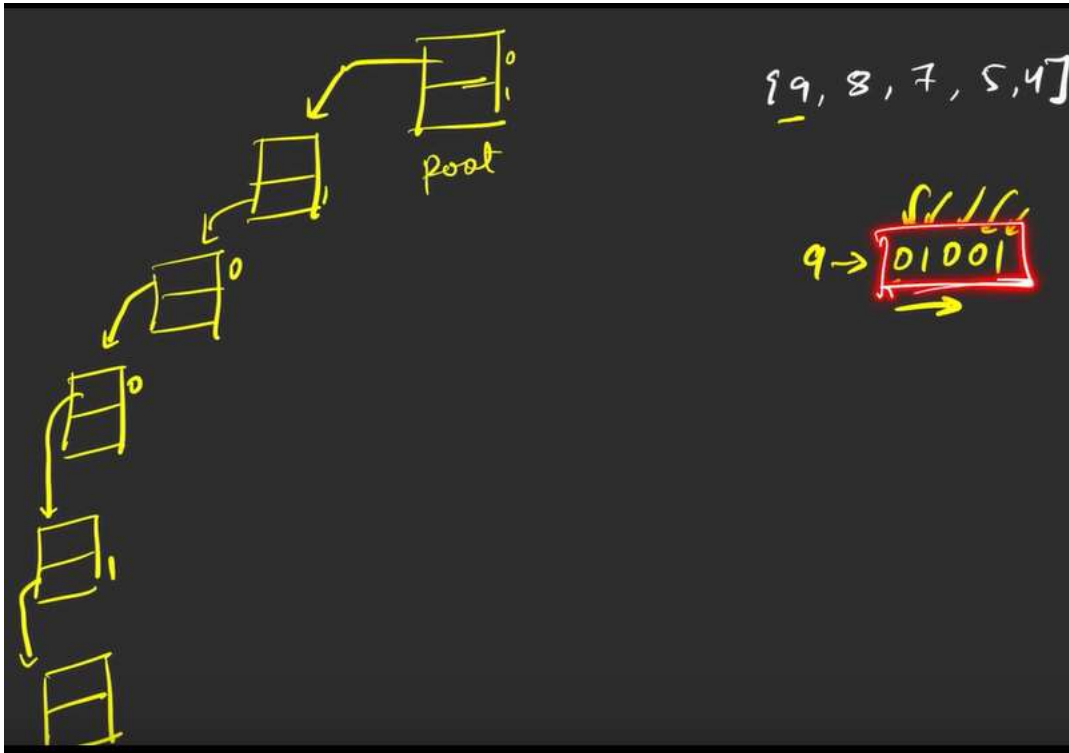
We pick "9" and store it in binary form, starting from leftmost element

9 -> 01001 (say)

We create a ref pointer for 0 in root

Now go to "1", make ref node

Now "0" then "0", now "0" now go to "1"

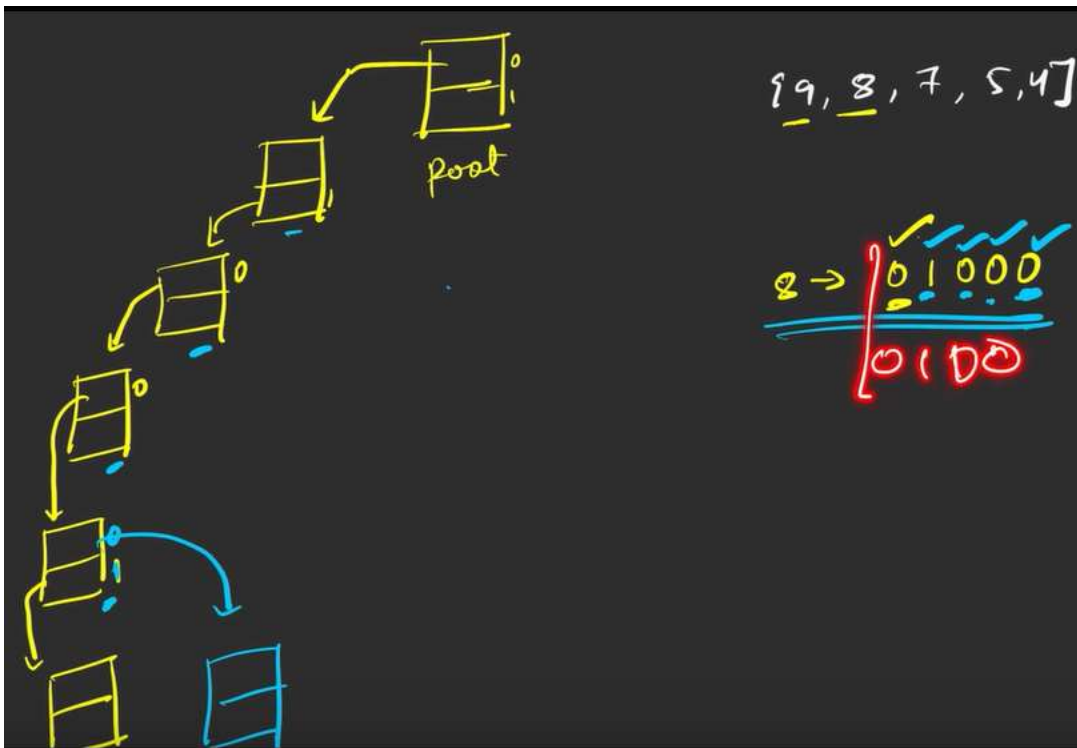


Now let us see "8" -> 01000

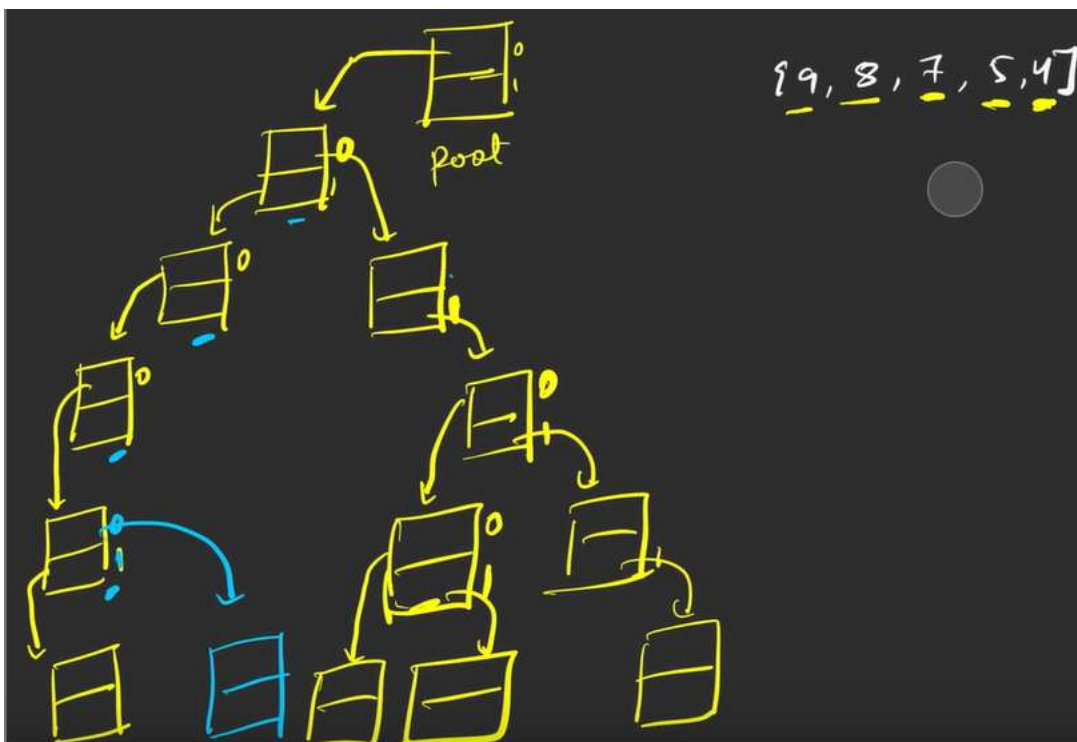
"0" is present so go to its ref node

"1" also present so go to its ref node

"0" also but "0" not exist so make one more ref node



Sameway we do for 7,5,4



So all elements are inserted into the Trie, successfully.  
Now let us take particular 8

### How to Maximize XOR

We need  $8 \oplus \text{arr}[i] = \text{maximum}$   
Say  $x = 8$  (01000)



We want the leftmost bits to be set/unset means if 8 ki 10th bit from left is 0 we need arr[i] ki 10th bit from left =1 so that  $0 \wedge 1 = 1$  and this way all bit will be reversed in such a way that we get XOR always 1 and we form a maximum number

In our question, we have  $x = 8$  (say), array = [9,8,7,5,4]

We check leftmost bit of 8 = 0, we check in all array element and leftmost bit of all is 0 so need to go with 0 anyhow

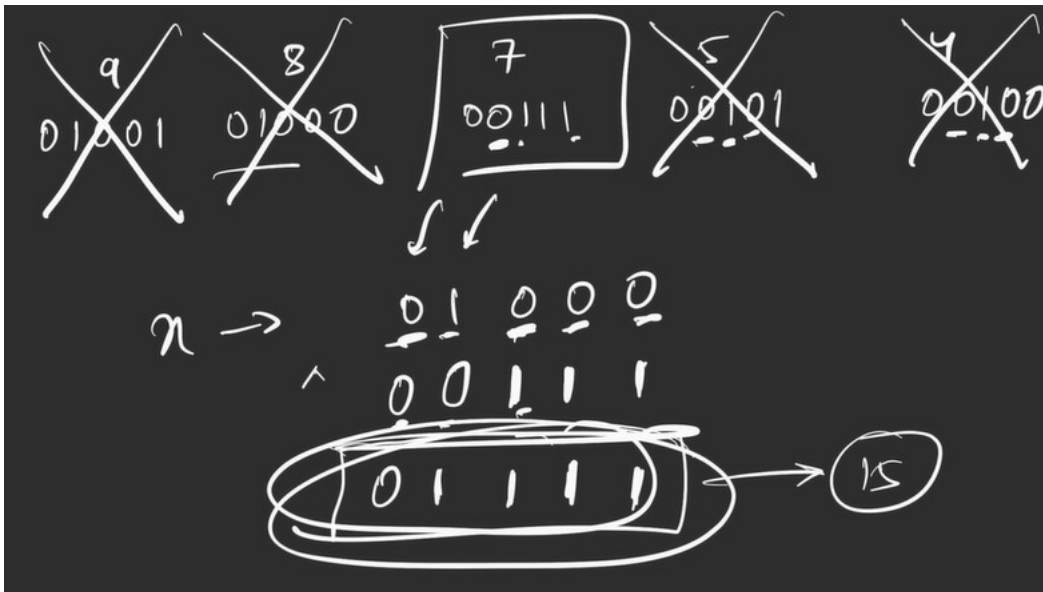
Now we move to next bit i.e 1 of "8"

We check in array, we need 0, so cannot take 9,8 but can take 7,5,4, now

Now we go to next bit of "8" i.e 0, we check 7,5,4 and to max XOR we need 1 so we check in 7,5,4. we can take 7,5,4 everyone

Now go to next of "8", we need 1 now so 7 is only one qualified and we check further.

If reversed bit does not exist in any answer, take what is there



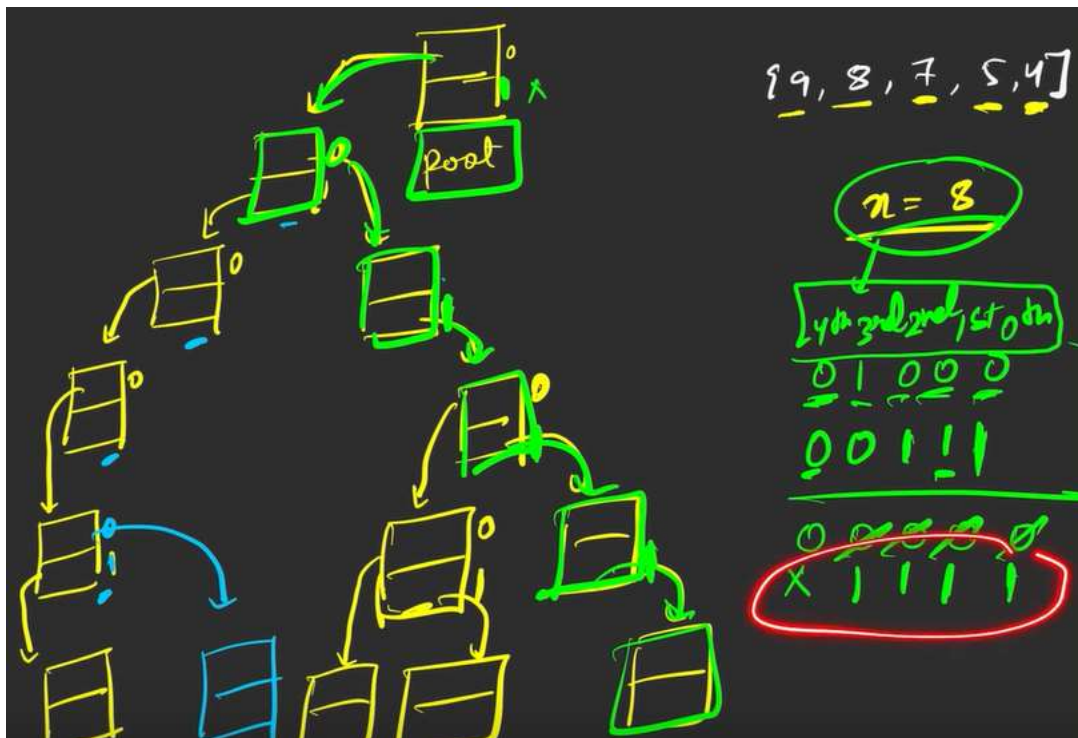
Initially we keep variable maxXOR = 00000000000000000000000000000000

Say  $x = 8$

We begin from leftmost of "8" we check in root of trie, do you have reverse of leftmost bit of "8", if not present take whatever we have in root and move to next leftmost bit of "8"

And so on.

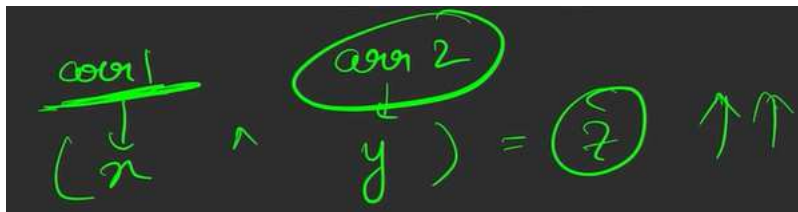
We keep turning on the bits



Make sure to insert 32 bits

## Original Problem

We need to pick  $x$  from  $arr1$  and  $y$  from  $arr2$  such that  $x \wedge y = \text{maximum}$



We insert  $arr1$  in Trie and we use all element in  $arr2$  as 0

We try  $arr2[0]$  with all elements of Trie

We try  $arr2[1]$  with all elements of Trie

We try  $arr2[2]$  with all elements of Trie

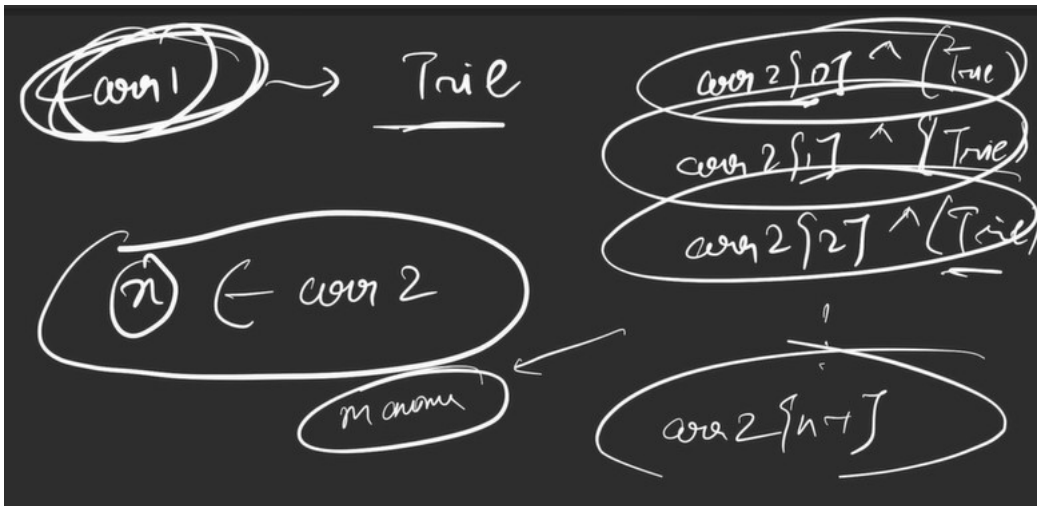
We try  $arr2[3]$  with all elements of Trie

We try  $arr2[4]$  with all elements of Trie

.... So on

We try  $arr2[n-1]$  with all elements of Trie

And whichever gives us maximum, we return it



## Our implementation of Trie class

We will have rootm insert(), getMax() which we call for every element of arr2

```
class Trie {
    root
    insert()
    getMax(n)
}
}
```

**TC:  $O(n*32)$  for arr1 insertion and  $O(m*32)$  for arr2**

```
struct Node {
    Node * links[2]; // just for 0 and 1
    bool containsKey(int ind) {
        return (links[ind] != NULL);
    }
    Node * get(int ind) {
        return links[ind];
    }
    void put(int ind, Node * node) {
        links[ind] = node;
    }
};
```

```

class Trie {
private: Node * root;
public:
    Trie() {
        root = new Node();
    }

public:
    void insert(int num) {
        Node * node = root;
        // start from leftmost bit so use reverse loop
        // leftmost bit is 31st bit so start loop from 31 till 0
        for (int i = 31; i >= 0; i--) {
            // below operation gives us ith bit
            int bit = (num >> i) & 1;
            if (!node->containsKey(bit)) {
                // if bit not present, insert it
                node->put(bit, new Node());
            }
            // go to ref node of that bit
            node = node->get(bit);
        }
    }

public:
    int findMax(int num) {
        // To find maximum XOR, for that we will use that reverse technique
        // Find the reverse of ith bit of num in trie to maximize the answer
        Node * node = root;
        int maxNum = 0; // it means 00000000000000000000000000000000
        for (int i = 31; i >= 0; i--) {
            // Get ith bit of num
            int bit = (num >> i) & 1;
            // instead of !
            // if bit = 1, then 1-0 = 1
            // if bit = 0, then 1-1 = 0
            if (node->containsKey(!bit)) {
                // if we have reverse bit in Trie store in maximum
                // 1<<i means Turn on (make it 1) that ith bit in maxNum as
                // our maxNum = 00000000000000000000000000000000 initially
                maxNum = maxNum | (1 << i);
                // get to ref node now
                node = node->get(!bit);
            } else {
                // if reverse bit not present, use whatever is there
                node = node->get(bit);
            }
        }
        return maxNum;
    }
};

```

```

int maxXOR(int n, int m, vector<int> &arr1, vector<int> &arr2)

```

```

{
    Trie trie;
    for (int i = 0; i < n; i++) {
        // insert whole arr1 in Trie
        trie.insert(arr1[i]);
    }
    int maxi = 0;
    for (int i = 0; i < m; i++) {
        // Now trie contains all node of arr1 so use getMax and find m
        // ax answer in Trie for arr2 ka har element
        maxi = max(maxi, trie.findMax(arr2[i]));
    }
    return maxi;
}

```

## Maximum XOR With an Element From Array

You are given an array/list 'ARR' consisting of 'N' non-negative integers. You are also given a list 'QUERIES' consisting of 'M' queries, where the 'i-th' query is a list/array of two non-negative integers 'Xi', 'Ai', i.e 'QUERIES[i]' = ['Xi', 'Ai'].

The answer to the ith query, i.e 'QUERIES[i]' is the maximum bitwise xor value of 'Xi' with any integer less than or equal to 'Ai' in 'ARR'.

You should return an array/list consisting of 'N' integers where the 'i-th' integer is the answer of 'QUERIES[i]'.

Note:

1. If all integers are greater than 'Ai' in array/list 'ARR' then the answer to this ith query will be -1.

**Sample Input 1:**

```

2
5 2
0 1 2 3 4
1 3
5 6
1 1
1
1 0

```

**Sample Output 1:**

```

3 7
-1

```

**Explanation of sample input 1:**

In the first test case, the answer of query [1, 3] is 3 because  $1^2 = 3$  and  $2 \leq 3$ , and the answer of query [5, 6] is 7 because  $5^2 = 7$  and  $2 \leq 6$ .

In the second test case, no element is less than or equal to 0 in the given array 'ARR'.

**Sample Input 2:**

```

2
6 3
6 6 3 5 2 4
6 3
8 1
12 4
5 2

```

0 0 0 0

1 0

1 1

Sample Output 2:

5 -1 15

1 1

## Approach

We know how to find  $\max(\text{arr}[i] \wedge x)$  problem from above

### Brute Force

```
for (i = 0 -> q-1)
{
    xi = queries[i][0]
    ai = queries[i][1]
    maxXOR = -1
    for (j = 0; j < n; j++)
    {
        if (arr[j] <= ai)
            maxXOR = max(maxXOR,
                          arr[j] ^ xi)
    }
}
ans.pb(maxXOR);
```

### Code:

```
vector < int > maxXorQueries(vector < int > & arr, vector < vector  
< int >> & queries) {  
    int n = arr.size();  
    int m = queries.size();  
    vector < int > ans(m, -1);  
    for (int i = 0; i < m; i++) {  
        for (int j = 0; j < n; j++) {  
            if (arr[j] <= queries[i][1]) {  
                ans[i] = max(ans[i], arr[j] ^ queries[i][0]);  
            }  
        }  
    }  
    return ans;  
}
```

}

## Optimised Approach

We will use a Trie,

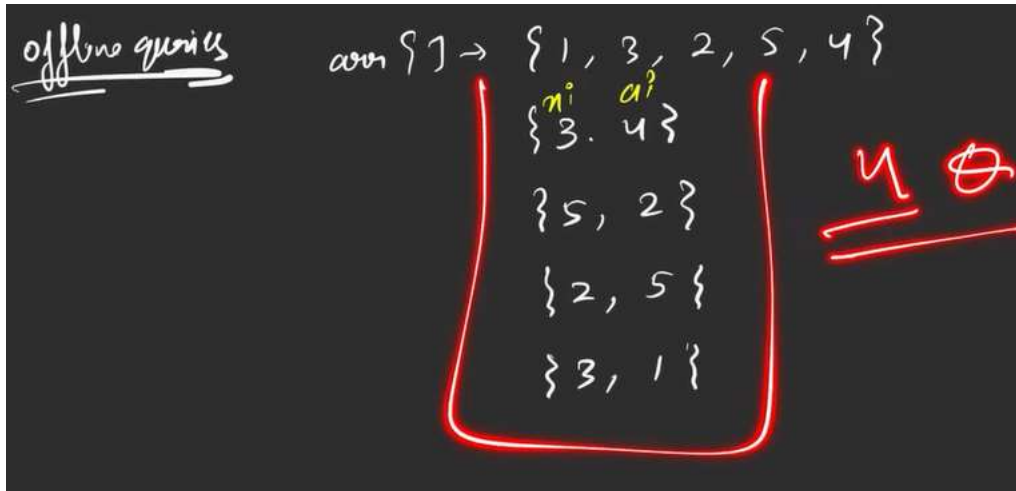
We need to give max (  $\text{arr}[i] \wedge x_i$  ) such that  $\text{arr}[i]$  is smaller than  $A_i$

What if we store only element  $< A_i$  in Trie?

So question becomes to find max XOR between  $\text{arr}[i]$  and  $x$

So to get  $A_i$ , there is a concept of **Offline Queries**.

We have been given an array and queries



Let say queries has 0-based indexing

What we do is, we solve for 0th query and store answer in  $\text{ans}[0]$

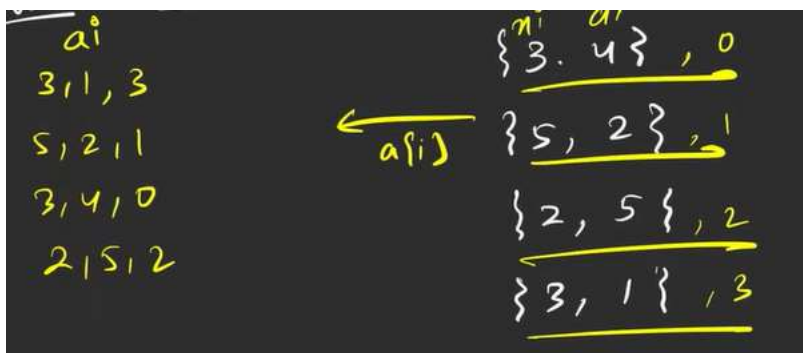
What we do is, we solve for 1st query and store answer in  $\text{ans}[1]$

What we do is, we solve for 2nd query and store answer in  $\text{ans}[2]$

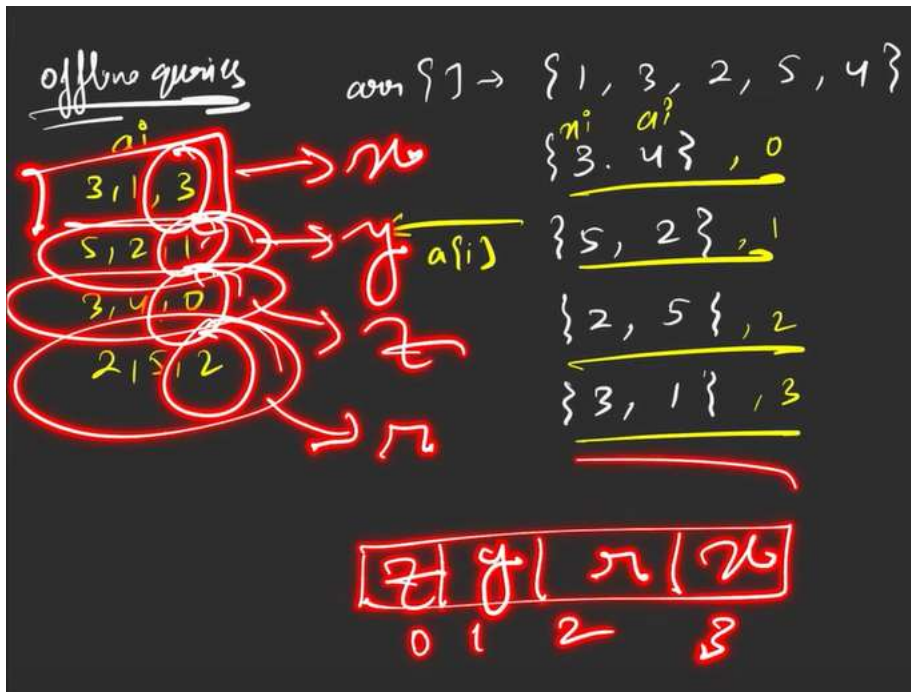
What we do is, we solve for 3rd query and store answer in  $\text{ans}[3]$

This is **online query solving**

We will sort according to  $A_i$  as  $A_i$  is of our need currently



We maintain an array of size = 4 and store answer according to query index which is solved



Here we are storing same as online query but we are maintaining a query number also, this is offline query

We will sort the arr[] given also



Now we pick query 1, which has  $A_i = 1$ , we go to arr we say first element  
 Are you  $\leq A_i$  i.e.  $\leq 1$ , Yes, store it in Trie  
 Go to next element, 2 are you  $\leq 1$ , No so do not insert it in Trie

$X_i = 3$  for query 1,  $A_i$  was 1 we do  $\text{getMax}(3)$  for whatever we have in Trie and store it in  $\text{ans}[3]$

Now solve for next element of query i.e. index = 1,  $A_i = 2$ ,  $X_i = 5$

We see 2 in arr[] is  $\leq 2$  so yes insert in trie  
 We see 3 in arr[] is  $\leq 2$  so stop

$\text{getMax}(5)$  for trie now which has 1,2 in it and store in  $\text{ans}[1]$



offline queries

$a_i$

3, 1, 3
5, 2, 1
3, 4, 0
2, 5, 2

$\rightarrow 2$

$L = 2$

arr  $\{ \} \rightarrow \{1, 3, 2, 5, 4\}$

$\{1, 2, 3, 4, 5\}$

True  $\rightarrow$

Now go for next query  $\{3, 4, 0\}$  where index = 0,  $X_i = 3$ ,  $A_i = 4$

offline queries

$a_i$

3, 1, 3
5, 2, 1
3, 4, 0
2, 5, 2

$\rightarrow ?$

$L = 4$

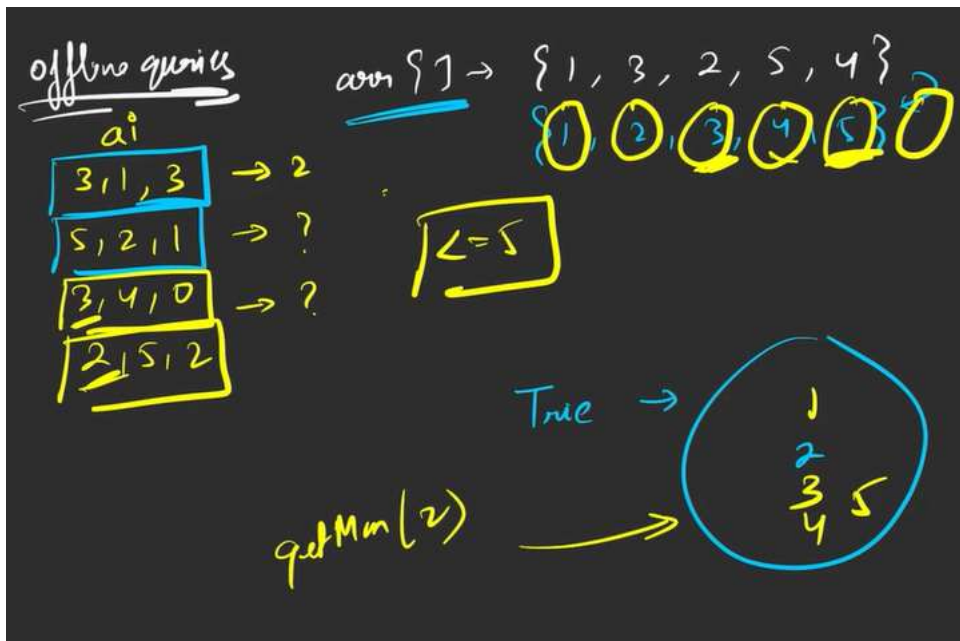
arr  $\{ \} \rightarrow \{1, 3, 2, 5, 4\}$

$\{1, 2, 3, 4, 5\}$

True  $\rightarrow$

query (3)

For  $\{2, 5, 2\}$  Query



So this way, we see and make sure, Trie only has element  $\leq A_i$  for each query

### Code:

```
struct Node {
    Node * links[2];
    bool containsKey(int ind) {
        return (links[ind] != NULL);
    }
    Node * get(int ind) {
        return links[ind];
    }
    void put(int ind, Node * node) {
        links[ind] = node;
    }
};

class Trie {
private: Node * root;
public:
    Trie() {
        root = new Node();
    }

public:
    void insert(int num) {
        Node * node = root;
        // cout << num << endl;
        for (int i = 31; i >= 0; i--) {
            int bit = (num >> i) & 1;
            if (!node->containsKey(bit)) {
                node->put(bit, new Node());
            }
            node = node->get(bit);
        }
    }
};
```

```

public:
    int findMax(int num) {
        Node * node = root;
        int maxNum = 0;
        for (int i = 31; i >= 0; i--) {
            int bit = (num >> i) & 1;
            if (node -> containsKey(!bit)) {
                maxNum = maxNum | (1 << i);
                node = node -> get(!bit);
            } else {
                node = node -> get(bit);
            }
        }
        return maxNum;
    }
};

vector<int> maxXorQueries(vector<int> &arr, vector<vector<int>> &queries) {
    vector<int> ans(queries.size(), 0);
    vector<pair<int, pair<int, int>>> offlineQueries;
    sort(arr.begin(), arr.end());
    int index = 0;
    for (auto & it: queries) {
        // store {Ai, Xi, index} we need to sort according to Ai so we store it like that
        offlineQueries.push_back({it[1], {it[0], index++}});
    }
    sort(offlineQueries.begin(), offlineQueries.end());
    int i = 0;
    int n = arr.size();
    Trie trie;
    for (auto & it: offlineQueries) {
        // find all indexes less than arr[i]
        while (i < n && arr[i] <= it.first) {
            // insert them all in trie
            trie.insert(arr[i]);
            i++;
        }
        // if we have inserted them successfully, store them accordingly to offline query index in our answer using findMax function we created in last problem
        if (i != 0) ans[it.second.second] = trie.findMax(it.second.first);
        // means no element found <= arr[i] so store -1 in answer
        else ans[it.second.second] = -1;
    }
    return ans;
}

```

## Power Set

Given a string S, Find all the possible subsequences of the String in lexicographically-sorted order.

**Example 1:**

**Input :** str = "abc"

**Output:** a ab abc ac b bc c

**Explanation :** There are 7 subsequences that can be formed from abc.

**Example 2:**

**Input:** str = "aa"

**Output:** a a aa

**Explanation :** There are 3 subsequences that can be formed from aa.

### Approach

$S = "abc" \rightarrow "", a, b, c, ab, ac, bc, abc$

Number of substring can be  $2^n$  where n is length of string

**How to check If ith bit is set or not**

Q) y  $i^{th}$  bit is set or not?

$$(n \& (1 << i)) \neq 0$$

We know there are for "abc", at max  $2^3 = 8$  subsequences

So we write 0 to 7 numbers with their bit representation

We know in bit rep, numbering is starts from 0 (rightmost) to leftmost bit

$S = "abc" \rightarrow "", a, b, c, ab, ac, bc, abc$

num	2	1	0 ← ind
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

Let say our string has indexing in this way

0 1 2  
S = "abc"

Now

0 -> we do not pick

1 -> we pick

Now start from bit rep of all number from 0 to 7

lower Set

S = "abc" → "", a, b, c

num	2	1	0	← ind	
0	0	0	0		→ ""
1	0	0	1		→ a
2	0	1	0		→ b
3	0	1	1		→ ab
4	1	0	0		→ c
5	1	0	1		→ ac
6	1	1	0		→ bc
7	1	1	1		→ abc

Code

```

fun( num = (0 → 2^n - 1)
{
    sub = "";
    fun( i = (0 - n - 1)
    {
        if (num & (1 << i))
            sub += s[i];
    }
    print(sub)
}

```

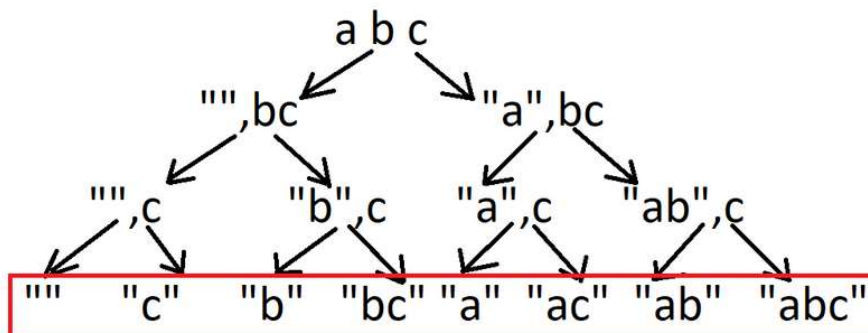
**TC:  $O(2^n * n)$**

**SC:  $O(1)$**

```
vector<string> AllPossibleStrings(string s) {
    int n = s.length();
    vector<string>ans;
    for (int num = 0; num < (1 << n); num++) {
        string sub = "";
        for (int i = 0; i < n; i++) {
            //check if the ith bit is set or not
            if (num & (1 << i)) {
                sub += s[i];
            }
        }
        if (sub.length() > 0) {
            ans.push_back(sub);
        }
    }
    sort(ans.begin(), ans.end());
    return ans;
}
```

## Using Recursion/ backtracking

**Approach:**



Since we are generating subsets two cases will be possible, either you can pick the character or you cannot pick the character and move to the next character.

### Approach

- Maintain a temp string (say f), which is empty initially.
- Now you have two options, either you can pick the character or not pick the character and move to the next index.
- Firstly we pick the character at ith index and then move to the next index. (f + s[i])
- If the base condition is hit, i.e. i == s.length(), then we print the temp string and return.
- Now while backtracking we have to pop the last character since now we have to implement the non-pick condition and then move to next index.

**Time Complexity:**  $O(2^n)$

**Space Complexity:**  $O(n)$ , recursion stack.

## Code

```
void solve(int i, string s, string &f) {
    if (i == s.length()) {
        cout << f << " ";
        return;
    }
    //picking
    f = f + s[i];
    solve(i + 1, s, f);
    //popping out while backtracking
    f.pop_back();
    solve(i + 1, s, f);
}
```