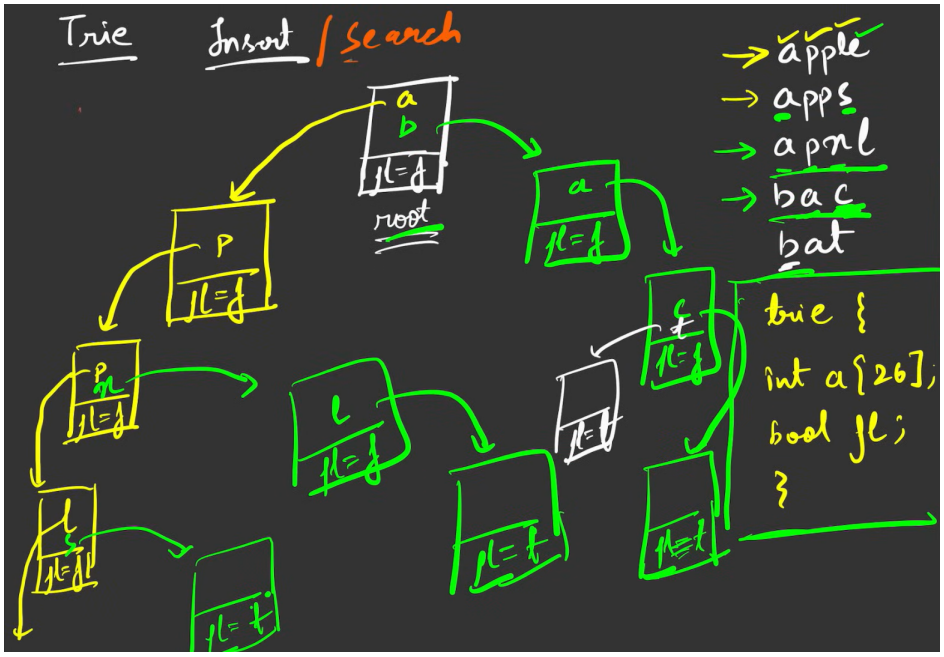
Tries :

- Allows operations like insert, search and startswith




```
class TrieNode {
public:
    unordered_map<char, TrieNode*> children;
    bool isEndOfWord;

    TrieNode() {
        isEndOfWord = false;
    }
};

class Trie {
private:
    TrieNode* root;

public:
    Trie() {
        root = new TrieNode();
    }

    // Insert a word into the trie
    void insert(string word) {
        TrieNode* current = root;
        for (char c : word) {
```

```
            if (current->children.find(c) == current->children.end()) {
                current->children[c] = new TrieNode();
            }
            current = current->children[c];
        }
        current->isEndOfWord = true;
    }

    // Search for a word in the trie
    bool search(string word) {
        TrieNode* current = root;
        for (char c : word) {
            if (current->children.find(c) == current->children.end()) {
                return false;
            }
            current = current->children[c];
        }
        return current->isEndOfWord;
    }

    // Check if there is any word in the trie that starts with the given prefix
    bool startsWith(string prefix) {
        TrieNode* current = root;
        for (char c : prefix) {
            if (current->children.find(c) == current->children.end()) {
                return false;
            }
            current = current->children[c];
        }
        return true;
    }
};
```

- When say there are two words app and apple → the node which has map from **l → e** (will have its isEndOfWord = True)

Say we are allowed to add duplicate words and even delete a word then ?

```
class TrieNode {
public:
    unordered_map<char, TrieNode*> children;
    int wordCount;   // Number of words ending at this node
    int prefixCount; // Number of words passing through this node

    TrieNode() {
        wordCount = 0;
        prefixCount = 0;
    }
```

```
};

class Trie {
private:
    TrieNode* root;

public:
    Trie() {
        root = new TrieNode();
    }

    // Insert a word into the trie
    void insert(string word) {
        TrieNode* current = root;
        for (char c : word) {
            if (current->children.find(c) == current->children.end()) {
                current->children[c] = new TrieNode();
            }
            current = current->children[c];
            current->prefixCount++;
        }
        current->wordCount++;
    }

    // Count the number of words equal to the given word
    int countWordsEqualTo(string word) {
        TrieNode* current = root;
        for (char c : word) {
            if (current->children.find(c) == current->children.end()) {
                return 0;
            }
            current = current->children[c];
        }
        return current->wordCount;
    }

    // Count the number of words starting with the given prefix
    int countWordsStartingWith(string prefix) {
        TrieNode* current = root;
        for (char c : prefix) {
            if (current->children.find(c) == current->children.end()) {
                return 0;
            }
            current = current->children[c];
        }
        return current->prefixCount;
    }
```

```
    // Erase a word from the trie
    void erase(string word) {
        if (countWordsEqualTo(word) == 0) return;
        // If the word doesn't exist, return

        TrieNode* current = root;
        for (char c : word) {
            if (current->children.find(c) != current->children.end()) {
                current = current->children[c];
                current->prefixCount--;
            }
        }
        current->wordCount--;
    }
};
```

Longest Word With All Prefixes

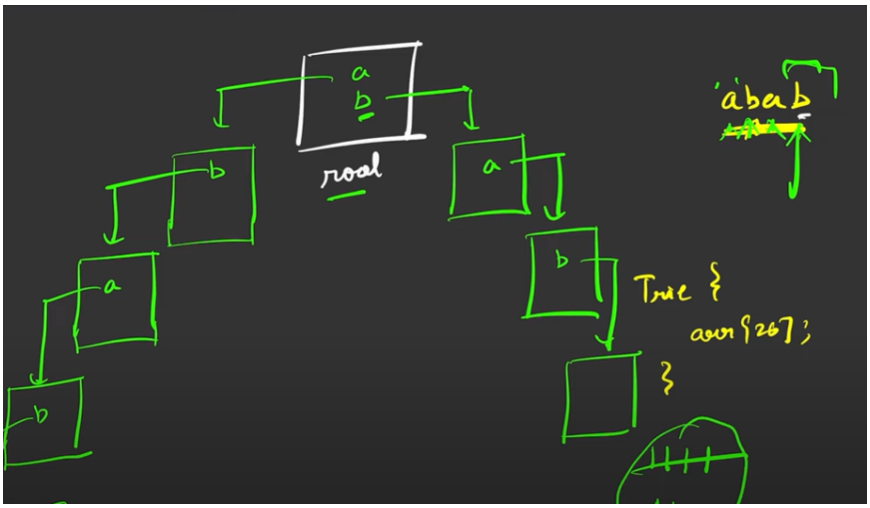
```
["a", "ap", "app", "appl", "apple", "applet"]

root
 |
 a (isEndOfWord = true)
 |
 p (isEndOfWord = true)
 |
 p (isEndOfWord = true)
 |
 l (isEndOfWord = true)
 |
 e (isEndOfWord = true)
 |
 t (isEndOfWord = true)
```

```
class TrieNode {
public:
    unordered_map<char, TrieNode*> children;
    bool isEndOfWord;

    TrieNode() {
        isEndOfWord = false;
    }
};

class Trie {
```

```
private:
    TrieNode* root;

    bool allPrefixesExist(const string& word) {
        TrieNode* current = root;
        for (char c : word) {
            if (current->children.find(c) == current->children.end()
            || !current->children[c]->isEndOfWord) {
                return false;
            }
            current = current->children[c];
        }
        return true;
    }

public:
    Trie() {
        root = new TrieNode();
    }

    void insert(string word) {
        TrieNode* current = root;
        for (char c : word) {
            if (current->children.find(c) == current->children.end()) {
                current->children[c] = new TrieNode();
            }
            current = current->children[c];
        }
        current->isEndOfWord = true;
    }

    string findLongestWordWithAllPrefixes(const vector<string>& words) {
        string longestWord = "";
        for (const string& word : words) {
            insert(word);
        }

        for (const string& word : words) {
            if (allPrefixesExist(word)) {
                if (word.length() > longestWord.length() ||
                (word.length() == longestWord.length() && word < longestWord)) {
                    longestWord = word;
                }
            }
        }

        return longestWord;
```

```
    }
};
```

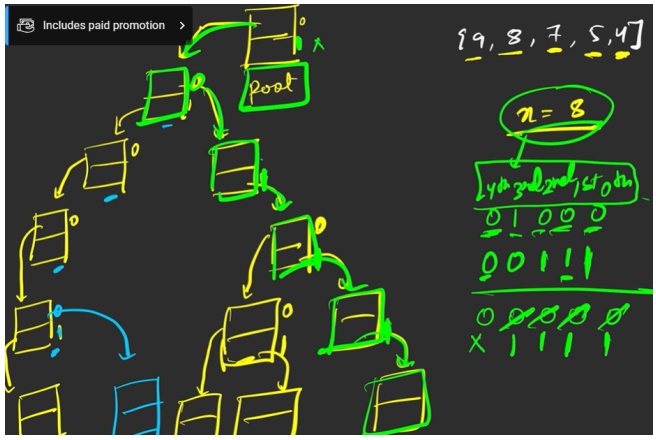
Number of Distinct Substrings in a String (No Duplicates)




```
class TrieNode {
public:
    unordered_map<char, TrieNode*> children;
};

class Trie {
private:
    TrieNode* root;

public:
    Trie() {
        root = new TrieNode();
    }

    void insertSuffix(const string& suffix) {
        TrieNode* current = root;
        for (char c : suffix) {
            if (current->children.find(c) == current->children.end()) {
                current->children[c] = new TrieNode();
            }
            current = current->children[c];
        }
    }

    int countNodes(TrieNode* node) {
        int count = 0;
        for (auto it : node->children) {
```

```
            count += countNodes(it.second);
        }
        return 1 + count;// Count this node + all its children
    }

    int countDistinctSubstrings(const string& s) {
        for (int i = 0; i < s.length(); i++) {
            insertSuffix(s.substr(i));
        }
        return countNodes(root) - 1; // Subtract 1 to exclude the root itself
    }
};
```

Maximum XOR of Two Numbers in an Array




```
class TrieNode {
public:
    TrieNode* children[2];

    TrieNode() {
        children[0] = nullptr;
        children[1] = nullptr;
    }
};

class Trie {
private:
    TrieNode* root;

public:
    Trie() {
        root = new TrieNode();
    }
```

```
    // Insert number into Trie
    void insert(int num) {
        TrieNode* node = root;
        for (int i = 31; i >= 0; --i) {
        // 32 bits for standard integers (MSB - LSB)
            int bit = (num >> i) & 1;
            if (node->children[bit] == nullptr) {
                node->children[bit] = new TrieNode();
            }
            node = node->children[bit];
        }
    }

    // Find maximum XOR for given number
    int findMaxXor(int num) {
        TrieNode* node = root;
        int maxXor = 0;
        for (int i = 31; i >= 0; --i) {  // 32 bits for standard integers
            int bit = (num >> i) & 1;
            int toggleBit = 1 - bit; // So that XOR would result in 1 :)
            if (node->children[toggleBit] != nullptr) {
                maxXor = (maxXor << 1) | 1;
                node = node->children[toggleBit];
            } else {
                maxXor = (maxXor << 1);
                node = node->children[bit];
            }
        }
        return maxXor;
    }
};

int findMaximumXor(vector<int>& nums) {
    Trie trie;
    int maxXor = 0;
    for (int num : nums) {
        trie.insert(num);
    }
    for (int num : nums) {
        maxXor = max(maxXor, trie.findMaxXor(num));
    }
    return maxXor;
}
```

- **Online queries** are queries that need to be answered immediately as they are received. The system must process each query in real-time, without knowledge of future queries.

- **Offline queries** are queries that are processed after all queries have been collected. The system can use the full set of queries to optimize the processing and make decisions based on all available data.

Maximum XOR With an Element From Array

Given an array.

Each time we are given a query (xi, ai). Return the max XOR of xi with elements in array (Consider only those ≤ ai).

- Offline queries → sort them based on ai. As ai increases build the tries, compute the previous algo and keep continuing