**01.) IMPLEMENT TRIE**

struct Node{

    Node \*links[26];

    bool flag=false;

    bool containsKey(char ch){

        return links[ch-'a']!=NULL;

    }

    void put(char ch, Node\* node){

        links[ch-'a']=node;

    }

    Node\* get(char ch){

        return links[ch-'a'];

    }

    void setEnd(){

        flag=true;

    }

    bool isEnd(){

        return flag;

    }

};

class Trie {

private:

    Node\* root;

public:

    Trie() {

        root=new Node();

    }

    void insert(string word) {

        Node\* node=root;

        for(int i=0;i<word.length();i++){

            if(!node->containsKey(word[i])){

                node->put(word[i], new Node());

            }

            node=node->get(word[i]);

        }

        node->setEnd();

    }

    bool search(string word) {

        Node\* node=root;

        for(int i=0;i<word.length();i++){

            if(!node->containsKey(word[i])){

                return false;

            }

            node=node->get(word[i]);

        }

        return node->isEnd();

    }

    bool startsWith(string prefix) {

        Node\* node=root;

        for(int i=0;i<prefix.length();i++){

            if(!node->containsKey(prefix[i])){

                return false;

            }

            node=node->get(prefix[i]);

        }

        return true;

    }

};

/\*\*

 \* Your Trie object will be instantiated and called as such:

 \* Trie\* obj = new Trie();

 \* obj->insert(word);

 \* bool param\_2 = obj->search(word);

 \* bool param\_3 = obj->startsWith(prefix);

 \*/

**02.) IMPLEMENT TRIE II**

#include <bits/stdc++.h>

struct Node{

    Node\* links[26];

    int cntEndWith=0;

    int cntPrefix=0;

    bool containsKey(char ch){

        return (links[ch-'a']!=NULL);

    }

    Node\* get(char ch){

        return links[ch-'a'];

    }

    void put(char ch, Node\* node){

        links[ch-'a']=node;

    }

    void increaseEnd(){

        cntEndWith++;

    }

    void increasePrefix(){

        cntPrefix++;

    }

    void deleteEnd(){

        cntEndWith--;

    }

    void reducePrefix(){

        cntPrefix--;

    }

    int getEnd(){

        return cntEndWith;

    }

    int getPrefix(){

        return cntPrefix;

    }

};

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->containsKey(word[i])){

                node->put(word[i], new Node());

            }

            node=node->get(word[i]);

            node->increasePrefix();

        }

        node->increaseEnd();

    }

    int countWordsEqualTo(string &word){

        Node \*node=root;

        for(int i=0;i<word.length();i++){

            if(node->containsKey(word[i])){

                node=node->get(word[i]);

            }

            else{

                return 0;

            }

        }

        return node->getEnd();

    }

    int countWordsStartingWith(string &word){

        Node\* node=root;

        for(int i=0;i<word.size();i++){

            if(node->containsKey(word[i])){

                node=node->get(word[i]);

            }

            else{

                return 0;

            }

        }

        return node->getPrefix();

    }

    void erase(string &word){

        Node\* node=root;

        for(int i=0;i<word.length();i++){

            if(node->containsKey(word[i])){

                node=node->get(word[i]);

                node->reducePrefix();

            }

            else{

                return;

            }

        }

        node->deleteEnd();

    }

};

**03.) COMPLETE STRING (LONGEST WORD WITH ALL PREFIXES)**

#include <bits/stdc++.h>

struct Node{

    Node\* links[26];

    bool flag=false;

    bool containsKey(char ch){

        return links[ch-'a'];

    }

    Node\* get(char ch){

        return links[ch-'a'];

    }

    void put(char ch, Node\* node){

        links[ch-'a']=node;

    }

    void setEnd(){

        flag=true;

    }

    bool isEnd(){

        return flag;

    }

};

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->containsKey(word[i])){

                node->put(word[i], new Node());

            }

            node=node->get(word[i]);

        }

        node->setEnd();

    }

    bool checkIfPrefixExists(string word){

        bool fl=true;

        Node\* node=root;

        for(int i=0;i<word.length();i++){

            if(node->containsKey(word[i])){

                node=node->get(word[i]);

                if(node->isEnd()==false)

                return false;

            }

            else{

                return false;

            }

        }

        return true;

    }

};

string completeString(int n, vector<string> &a){

    Trie trie;

    for(auto &it:a){

        trie.insert(it);

    }

    string longest="";

    for(auto &it:a){

        if(trie.checkIfPrefixExists(it)){

            if(it.length()>longest.length()){

                longest=it;

            }

            else if(it.length()==longest.length() && it<longest){

                longest=it;

            }

        }

    }

    if(longest=="")

    return "None";

    return longest;

}

**04.) COUNT DISTINCT SUBSTRINGS**

class Node{

    Node\* links[26];

    public:

    bool containsKey(char ch){

        return links[ch-'a']!=NULL;

    }

    void put(char ch, Node\* node){

        links[ch-'a']=node;

    }

    Node\* get(char ch){

        return links[ch-'a'];

    }

};

int countDistinctSubstrings(string &word)

{

    int cnt=0;

    Node\* root=new Node();

    for(int i=0;i<word.size();i++){

        Node\* node=root;

        for(int j=i;j<word.size();j++){

            if(!node->containsKey(word[j])){

                cnt++;

                node->put(word[j], new Node());

            }

            node=node->get(word[j]);

        }

    }

    return cnt+1;

}

**05.) MAXIMUM XOR**

#include <iostream>

#include <vector>

using namespace std;

class TrieNode {

public:

    TrieNode\* children[2];

    TrieNode() {

        children[0] = nullptr;

        children[1] = nullptr;

    }

};

void insert(TrieNode\* root, int num) {

    TrieNode\* curr = root;

    for (int i = 31; i >= 0; i--) {

        int bit = (num >> i) & 1;

        if (!curr->children[bit]) {

            curr->children[bit] = new TrieNode();

        }

        curr = curr->children[bit];

    }

}

int findMaxXOR(TrieNode\* root, vector<int>& nums) {

    int maxXOR = 0;

    for (int num : nums) {

        TrieNode\* curr = root;

        int currXOR = 0;

        for (int i = 31; i >= 0; i--) {

            int bit = (num >> i) & 1;

            if (curr->children[1 - bit]) {

                currXOR |= (1 << i);

                curr = curr->children[1 - bit];

            } else {

                curr = curr->children[bit];

            }

        }

        maxXOR = max(maxXOR, currXOR);

    }

    return maxXOR;

}

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

    TrieNode\* root = new TrieNode();

    for (int num : arr1) {

        insert(root, num);

    }

    return findMaxXOR(root, arr2);

}

**06.) MAXIMUM XOR OF TWO NUMBER 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();

    }

    void insert(int num) {

        TrieNode\* node = root;

        for (int i = 31; i >= 0; i--) {

            int bit = (num >> i) & 1;

            if (!node->children[bit]) {

                node->children[bit] = new TrieNode();

            }

            node = node->children[bit];

        }

    }

    int getMaxXOR(int num) {

        TrieNode\* node = root;

        int maxXOR = 0;

        for (int i = 31; i >= 0; i--) {

            int bit = (num >> i) & 1;

            if (node->children[1 - bit]) {

                maxXOR = maxXOR | (1 << i);

                node = node->children[1 - bit];

            } else {

                node = node->children[bit];

            }

        }

        return maxXOR;

    }

};

class Solution {

public:

    int findMaximumXOR(vector<int>& nums) {

        Trie trie;

        for (int num : nums) {

            trie.insert(num);

        }

        int maxXOR = 0;

        for (int num : nums) {

            maxXOR = max(maxXOR, trie.getMaxXOR(num));

        }

        return maxXOR;

    }

};

**07.) MAXIMUM XOR WITH AN ELEMENT IN ARRAY**

class TrieNode {

public:

    TrieNode\* children[2];

    TrieNode() {

        children[0] = nullptr;

        children[1] = nullptr;

    }

};

class Trie {

private:

    TrieNode\* root;

public:

    Trie() {

        root = new TrieNode();

    }

    void insert(int num) {

        TrieNode\* node = root;

        for (int i = 31; i >= 0; i--) {

            int bit = (num >> i) & 1;

            if (!node->children[bit]) {

                node->children[bit] = new TrieNode();

            }

            node = node->children[bit];

        }

    }

    int getMaxXOR(int num) {

        TrieNode\* node = root;

        int maxXOR = 0;

        for (int i = 31; i >= 0; i--) {

            int bit = (num >> i) & 1;

            if (node->children[1 - bit]) {

                maxXOR = maxXOR | (1 << i);

                node = node->children[1 - bit];

            } else {

                node = node->children[bit];

            }

        }

        return maxXOR;

    }

};

class Solution {

public:

    vector<int> maximizeXor(vector<int>& nums, vector<vector<int>>& queries) {

        sort(nums.begin(), nums.end());

        vector<int> result(queries.size(), -1);

        vector<pair<int, pair<int, int>>> offlineQueries;

        for (int i = 0; i < queries.size(); i++) {

            offlineQueries.push\_back({queries[i][1], {queries[i][0], i}});

        }

        sort(offlineQueries.begin(), offlineQueries.end());

        Trie trie;

        int index = 0;

        for (auto& q : offlineQueries) {

            int m = q.first;

            int x = q.second.first;

            int queryIndex = q.second.second;

            while (index < nums.size() && nums[index] <= m) {

                trie.insert(nums[index]);

                index++;

            }

            if (index > 0) {

                result[queryIndex] = trie.getMaxXOR(x);

            }

        }

        return result;

    }

};

**08.) MAXIMUM GENETIC DIFFERENCE QUERY**

class TrieNode {

public:

    TrieNode\* children[2];

    int count;

    TrieNode() {

        children[0] = nullptr;

        children[1] = nullptr;

        count = 0;

    }

};

class Trie {

private:

    TrieNode\* root;

public:

    Trie() {

        root = new TrieNode();

    }

    void insert(int num) {

        TrieNode\* node = root;

        for (int i = 31; i >= 0; i--) {

            int bit = (num >> i) & 1;

            if (!node->children[bit]) {

                node->children[bit] = new TrieNode();

            }

            node = node->children[bit];

            node->count++;

        }

    }

    void remove(int num) {

        TrieNode\* node = root;

        for (int i = 31; i >= 0; i--) {

            int bit = (num >> i) & 1;

            node = node->children[bit];

            node->count--;

        }

    }

    int getMaxXOR(int num) {

        TrieNode\* node = root;

        int maxXOR = 0;

        for (int i = 31; i >= 0; i--) {

            int bit = (num >> i) & 1;

            if (node->children[1 - bit]

&& node->children[1 - bit]->count > 0) {

                maxXOR |= (1 << i);

                node = node->children[1 - bit];

            } else {

                node = node->children[bit];

            }

        }

        return maxXOR;

    }

};

class Solution {

public:

    void dfs(int node, vector<vector<int>>& adj,

vector<vector<pair<int, int>>>& nodeQueries,

vector<int>& result, Trie& trie) {

        trie.insert(node);

        for (auto& query : nodeQueries[node]) {

            int val = query.first;

            int idx = query.second;

            result[idx] = trie.getMaxXOR(val);

        }

        for (int child : adj[node]) {

            dfs(child, adj, nodeQueries, result, trie);

        }

        trie.remove(node);

    }

    vector<int> maxGeneticDifference(vector<int>& parents,

vector<vector<int>>& queries) {

        int n = parents.size();

        vector<vector<int>> adj(n);

        vector<vector<pair<int, int>>> nodeQueries(n);

        int root = -1;

        for (int i = 0; i < n; i++) {

            if (parents[i] == -1) {

                root = i;

            } else {

                adj[parents[i]].push\_back(i);

            }

        }

        for (int i = 0; i < queries.size(); i++) {

            int node = queries[i][0];

            int val = queries[i][1];

            nodeQueries[node].emplace\_back(val, i);

        }

        vector<int> result(queries.size());

        Trie trie;

        dfs(root, adj, nodeQueries, result, trie);

        return result;

    }

};

**09.) MAXIMUM STRONG PAIR XOR II**

class TrieNode {

public:

    TrieNode\* children[2];

    int count;

    TrieNode() {

        children[0] = nullptr;

        children[1] = nullptr;

        count = 0;

    }

};

class Trie {

private:

    TrieNode\* root;

public:

    Trie() {

        root = new TrieNode();

    }

    void insertNode(int num) {

        TrieNode\* tmp = root;

        for (int i = 20; i >= 0; i--) {

            int ind = (num & (1 << i)) > 0 ? 1 : 0;

            if (tmp->children[ind] == nullptr) {

                tmp->children[ind] = new TrieNode();

            }

            tmp = tmp->children[ind];

            tmp->count++;

        }

    }

    int getMax(int num) {

        TrieNode\* tmp = root;

        int maxVal = 0;

        for (int i = 20; i >= 0; i--) {

            int ind = (num & (1 << i)) > 0 ? 1 : 0;

            int opp = ind == 0 ? 1 : 0;

            if (tmp->children[opp] != nullptr && tmp->children[opp]->count > 0) {

                maxVal |= (1 << i);

                tmp = tmp->children[opp];

            } else {

                tmp = tmp->children[ind];

            }

        }

        return maxVal;

    }

    void deleteNode(int num) {

        TrieNode\* tmp = root;

        for (int i = 20; i >= 0; i--) {

            int ind = (num & (1 << i)) > 0 ? 1 : 0;

            tmp = tmp->children[ind];

            tmp->count--;

        }

    }

};

class Solution {

public:

    int maximumStrongPairXor(vector<int>& nums) {

        int res = 0;

        Trie trie;

        sort(nums.begin(), nums.end());

        int i = 0;

        for (int num : nums) {

            while (i < nums.size() && nums[i] <= 2 \* num)

                trie.insertNode(nums[i++]);

            res = max(res, trie.getMax(num));

            trie.deleteNode(num);

        }

        return res;

    }

};

**10.) MAXIMUM STRONG PAIR XOR I**

**TRIE IMPLEMENTATION**

class TrieNode {

public:

    TrieNode\* children[2];

    int count;

    TrieNode() {

        children[0] = nullptr;

        children[1] = nullptr;

        count = 0;

    }

};

class Trie {

private:

    TrieNode\* root;

public:

    Trie() {

        root = new TrieNode();

    }

    void insertNode(int num) {

        TrieNode\* tmp = root;

        for (int i = 20; i >= 0; i--) {

            int ind = (num & (1 << i)) > 0 ? 1 : 0;

            if (tmp->children[ind] == nullptr) {

                tmp->children[ind] = new TrieNode();

            }

            tmp = tmp->children[ind];

            tmp->count++;

        }

    }

    int getMax(int num) {

        TrieNode\* tmp = root;

        int maxVal = 0;

        for (int i = 20; i >= 0; i--) {

            int ind = (num & (1 << i)) > 0 ? 1 : 0;

            int opp = ind == 0 ? 1 : 0;

            if (tmp->children[opp] != nullptr

&& tmp->children[opp]->count > 0) {

                maxVal |= (1 << i);

                tmp = tmp->children[opp];

            } else {

                tmp = tmp->children[ind];

            }

        }

        return maxVal;

    }

    void deleteNode(int num) {

        TrieNode\* tmp = root;

        for (int i = 20; i >= 0; i--) {

            int ind = (num & (1 << i)) > 0 ? 1 : 0;

            tmp = tmp->children[ind];

            tmp->count--;

        }

    }

};

class Solution {

public:

    int maximumStrongPairXor(vector<int>& nums) {

        int res = 0;

        Trie trie;

        sort(nums.begin(), nums.end());

        int i = 0;

        for (int num : nums) {

            while (i < nums.size() && nums[i] <= 2 \* num)

                trie.insertNode(nums[i++]);

            res = max(res, trie.getMax(num));

            trie.deleteNode(num);

        }

        return res;

    }

};

**NORMAL IMPLEMENTATION (CONSTRAINTS ARE VERY LESS)**

class Solution {

public:

    int maximumStrongPairXor(vector<int>& nums) {

        int maxXOR = 0;

        int n = nums.size();

        for (int i = 0; i < n; i++) {

            for (int j = i + 1; j < n; j++) {

                int strongPair = abs(nums[i] - nums[j]);

                int currentXOR = nums[i] ^ nums[j];

                if (strongPair <= min(nums[i], nums[j])) {

                    maxXOR = max(maxXOR, currentXOR);

                }

            }

        }

        return maxXOR;

    }

};

**11.) REPLACE WORDS**

class TrieNode {

public:

    unordered\_map<char, TrieNode\*> children;

    bool is\_end\_of\_word;

    TrieNode() : is\_end\_of\_word(false) {}

};

class Trie {

public:

    Trie() {

        root = new TrieNode();

    }

    void insert(const 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->is\_end\_of\_word = true;

    }

    string search\_shortest\_prefix(const string& word) {

        TrieNode\* current = root;

        string prefix = "";

        for (char c : word) {

            if (current->children.find(c) != current->children.end()) {

                current = current->children[c];

                prefix += c;

                if (current->is\_end\_of\_word) {

                    return prefix;

                }

            } else {

                break;

            }

        }

        return word;

    }

private:

    TrieNode\* root;

};

class Solution {

public:

    string replaceWords(vector<string>& dictionary, string sentence) {

        Trie trie;

        for (const string& word : dictionary) {

            trie.insert(word);

        }

        istringstream iss(sentence);

        string word;

        string result = "";

        while (iss >> word) {

            if (!result.empty()) {

                result += " ";

            }

            result += trie.search\_shortest\_prefix(word);

        }

        return result;

    }

};

**12.) WORD BREAK**

class TrieNode {

public:

    unordered\_map<char, TrieNode\*> children;

    bool is\_end\_of\_word;

    TrieNode() : is\_end\_of\_word(false) {}

};

class Trie {

public:

    Trie() {

        root = new TrieNode();

    }

    void insert(const 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->is\_end\_of\_word = true;

    }

    TrieNode\* getRoot() {

        return root;

    }

private:

    TrieNode\* root;

};

class Solution {

public:

    bool wordBreak(string s, vector<string>& wordDict) {

        Trie trie;

        for (const string& word : wordDict) {

            trie.insert(word);

        }

        int n = s.size();

        vector<bool> dp(n + 1, false);

        dp[0] = true;

        TrieNode\* root = trie.getRoot();

        for (int i = 0; i < n; ++i) {

            if (!dp[i]) continue;

            TrieNode\* current = root;

            for (int j = i; j < n; ++j) {

                char c = s[j];

                if (current->children.find(c) == current->children.end()) {

                    break;

                }

                current = current->children[c];

                if (current->is\_end\_of\_word) {

                    dp[j + 1] = true;

                }

            }

        }

        return dp[n];

    }

};

**13.) WORD BREAK II**

class TrieNode {

public:

    unordered\_map<char, TrieNode\*> children;

    bool is\_end\_of\_word;

    TrieNode() : is\_end\_of\_word(false) {}

};

class Trie {

public:

    Trie() {

        root = new TrieNode();

    }

    void insert(const 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->is\_end\_of\_word = true;

    }

    TrieNode\* getRoot() {

        return root;

    }

private:

    TrieNode\* root;

};

class Solution {

public:

    vector<string> wordBreak(string s, vector<string>& wordDict) {

        Trie trie;

        for (const string& word : wordDict) {

            trie.insert(word);

        }

        unordered\_map<int, vector<string>> memo;

        return wordBreakHelper(s, 0, trie.getRoot(), memo);

    }

private:

    vector<string> wordBreakHelper(const string& s, int start, TrieNode\* root,

unordered\_map<int, vector<string>>& memo) {

        if (memo.find(start) != memo.end()) {

            return memo[start];

        }

        vector<string> results;

        TrieNode\* current = root;

        string currentWord = "";

        for (int i = start; i < s.length(); ++i) {

            char c = s[i];

            if (current->children.find(c) == current->children.end()) {

                break;

            }

            current = current->children[c];

            currentWord += c;

            if (current->is\_end\_of\_word) {

                if (i == s.length() - 1) {

                    results.push\_back(currentWord);

                } else {

                    vector<string> subResults = wordBreakHelper(s, i + 1,

root, memo);

                    for (const string& subResult : subResults) {

                        results.push\_back(currentWord + " " + subResult);

                    }

                }

            }

        }

        memo[start] = results;

        return results;

    }

};

**14.) CONCATENATED WORDS**

class TrieNode {

public:

    unordered\_map<char, TrieNode\*> children;

    bool isEndOfWord;

    TrieNode() : isEndOfWord(false) {}

};

class Trie {

public:

    TrieNode\* root;

    Trie() {

        root = new TrieNode();

    }

    void insert(const 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;

    }

};

class Solution {

public:

    vector<string> findAllConcatenatedWordsInADict(vector<string>& words) {

        Trie trie;

        for (const string& word : words) {

            if (!word.empty()) {

                trie.insert(word);

            }

        }

        unordered\_set<string> wordSet(words.begin(), words.end());

        unordered\_map<string, bool> memo;

        vector<string> result;

        for (const string& word : words) {

            if (!word.empty() && canForm(word, trie.root, wordSet, memo)) {

                result.push\_back(word);

            }

        }

        return result;

    }

private:

    bool canForm(const string& word, TrieNode\* root,

const unordered\_set<string>& wordSet,

unordered\_map<string, bool>& memo) {

        if (memo.find(word) != memo.end()) {

            return memo[word];

        }

        TrieNode\* current = root;

        int n = word.size();

        for (int i = 0; i < n; ++i) {

            char c = word[i];

            if (current->children.find(c) == current->children.end()) {

                return memo[word] = false;

            }

            current = current->children[c];

            if (current->isEndOfWord && i != n - 1) {

                string suffix = word.substr(i + 1);

                if (wordSet.find(suffix) != wordSet.end() ||

canForm(suffix, root, wordSet, memo)) {

                    return memo[word] = true;

                }

            }

        }

        return memo[word] = false;

    }

};

**15.) EXTRA CHARACTERS IN A STRING**

class TrieNode {

public:

    unordered\_map<char, TrieNode\*> children;

    bool is\_end\_of\_word;

    TrieNode() : is\_end\_of\_word(false) {}

};

class Trie {

public:

    Trie() {

        root = new TrieNode();

    }

    void insert(const 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->is\_end\_of\_word = true;

    }

    TrieNode\* getRoot() {

        return root;

    }

private:

    TrieNode\* root;

};

class Solution {

public:

    int minExtraChar(string s, vector<string>& dictionary) {

        Trie trie;

        for (const string& word : dictionary) {

            trie.insert(word);

        }

        int n = s.size();

        vector<int> dp(n + 1, n);

        dp[0] = 0;

        TrieNode\* root = trie.getRoot();

        for (int i = 0; i < n; ++i) {

            TrieNode\* current = root;

            for (int j = i; j < n; ++j) {

                char c = s[j];

                if (current->children.find(c) == current->children.end()) {

                    break;

                }

                current = current->children[c];

                if (current->is\_end\_of\_word) {

                    dp[j + 1] = min(dp[j + 1], dp[i]);

                }

            }

            dp[i + 1] = min(dp[i + 1], dp[i] + 1);

        }

        return dp[n];

    }

};

**16.) Kth SMALLEST IN LEXICOGRAPHICAL ORDER**

class Solution {

public:

    int findKthNumber(int n, int k) {

        int current = 1;

        k--;

        while (k > 0) {

            int steps = calculateSteps(n, current, current + 1);

            if (steps <= k) {

                current++;

                k -= steps;

            } else {

                current \*= 10;

                k--;

            }

        }

        return current;

    }

private:

    int calculateSteps(int n, long long n1, long long n2) {

        int steps = 0;

        while (n1 <= n) {

            steps += min((long long)n + 1, n2) - n1;

            n1 \*= 10;

            n2 \*= 10;

        }

        return steps;

    }

};

**THANK YOU !**