**208. 实现 Trie (前缀树)**

实现一个 Trie (前缀树)，包含 insert, search, 和 startsWith 这三个操作。

Trie trie = new Trie();

trie.insert("apple");

trie.search("apple"); // 返回 true

trie.search("app"); // 返回 false

trie.startsWith("app"); // 返回 true

trie.insert("app");

trie.search("app"); // 返回 true

|  |
| --- |
| class **Trie** {  bool isEnd\_{ false };  vector<Trie\*> subNodes\_;  public:  **Trie()**  {  subNodes\_ = vector<Trie\*>(26, nullptr);  }  **~Trie()**  {  for (auto ptr : subNodes\_) {  delete ptr;  }  }  **void insert(string word)**  {  Trie\* cur = this;  for (char c : word) {  if (cur->subNodes\_[c - 'a'] == nullptr) {  cur->subNodes\_[c - 'a'] = new Trie();  }  cur = cur->subNodes\_[c - 'a'];  }  cur->isEnd\_ = true;  }  **bool search(string word)**  {  Trie\* cur = this;  for (auto c : word) {  if (cur->subNodes\_[c - 'a'] == nullptr) {  return false;  }  cur = cur->subNodes\_[c - 'a'];  }  return cur->isEnd\_;  }  **bool startsWith(string prefix)**  {  Trie\* cur = this;  for (auto c : prefix) {  if (cur->subNodes\_[c - 'a'] == nullptr) {  return false;  }  cur = cur->subNodes\_[c - 'a'];  }  return true;  }  }; |

**720. 词典中最长的单词**

给出一个字符串数组words组成的一本英语词典。从中找出最长的一个单词，该单词是由words词典中其他单词逐步添加一个字母组成。若其中有多个可行的答案，则返回答案中字典序最小的单词。

若无答案，则返回空字符串。

|  |
| --- |
| string longestWord(vector<string>& words) {  unordered\_set<string> dict;  sort(words.begin(),words.end(),[](const string& s1,const string& s2)  {return s1.size()<s2.size();});  for(const auto& w:words){  if(w.size()==1)  dict.insert(w);  else if(dict.find(w.substr(0,w.size()-1))!=dict.end())  dict.insert(w);  }  string cur;  for(const auto& d:dict){  if(d.size()==cur.size())  cur=min(cur,d);  else if(d.size()>cur.size())  cur=d;  }  return cur;  } |

|  |
| --- |
| class Solution {  vector<Trie\*> v\_{ 26 };  bool isEnd\_{false};  public:  string longestWord(vector<string>& words)  {  Trie tree;  for (const auto& word : words) {  tree.Insert(word);  }  sort(words.begin(), words.end());  string longest;  for (const auto& word : words) {  if (tree.Search(word) && word.size() > longest.size()) {  longest = word;  }  }  return longest;  }  private:  class Trie  {  public:  Trie() {}  virtual ~Trie(){}  void Insert(const string& word)  {  auto cur = this;  for (auto c : word) {  if (cur->v\_[c - 'a'] == nullptr) {  cur->v\_[c - 'a'] = new Trie();  }  cur = cur->v\_[c - 'a'];  }  cur->isEnd\_ = true;  }  bool Search(const string& word)  {  auto cur = this;  for (auto c : word) {  if (cur->v\_[c - 'a'] == nullptr) {  return false;  }  cur = cur->v\_[c - 'a'];  if (!cur->isEnd\_) {  return false;  }  }  return true;  }  };  }; |

**212. 单词搜索 II**

给定一个二维网格 board 和一个字典中的单词列表 words，找出所有同时在二维网格和字典中出现的单词。

单词必须按照字母顺序，通过相邻的单元格内的字母构成，其中“相邻”单元格是那些水平相邻或垂直相邻的单元格。同一个单元格内的字母在一个单词中不允许被重复使用。

输入:

words = ["oath","pea","eat","rain"] and board =

[

['o','a','a','n'],

['e','t','a','e'],

['i','h','k','r'],

['i','f','l','v']

]

输出: ["eat","oath"]

|  |
| --- |
| class Tire {  public:  Tire() {}  virtual ~Tire()  {  for (auto ptr : v\_) {  delete ptr;  }  }  void Insert(const string& word)  {  auto cur = this;  for (auto c : word) {  if (cur->v\_[c - 'a'] == nullptr) {  cur->v\_[c - 'a'] = new Tire();  }  cur = cur->v\_[c - 'a'];  }  cur->isEnd\_ = true;  }  Tire\* GetSub(char c) const  {  return v\_[c - 'a'];  }  bool IsEnd()  {  return isEnd\_;  }  private:  vector<Tire\*> v\_{ 26, nullptr };  bool isEnd\_{false};  };  class Solution {  public:  vector<string> findWords(vector<vector<char>>& board, vector<string>& words)  {  if (board.empty() || board[0].empty()) {  return {};  }  Tire tree;  for (const auto& word : words) {  tree.Insert(word);  max\_len = max(max\_len, (int)word.size());  }  n\_ = board.size();  m\_ = board[0].size();  for (size\_t y = 0; y < n\_; y++) {  for (size\_t x = 0; x < m\_; x++) {  Dfs(board, &tree, "", x, y);  }  }  return vector<string>(findWords\_.begin(), findWords\_.end());  }  void Dfs(vector<vector<char>>& board, const Tire\* tree, string str, int x, int y)  {  if (!IsValidPos(x, y) || tree == nullptr) {  return;  }  if (str.size() > max\_len) {  return;  }  char c = board[y][x];  if (c == '.') {  return;  }  str += c;  const auto next = tree->GetSub(c);  if (next != nullptr && next->IsEnd()) {  findWords\_.insert(str);  }  board[y][x] = '.';  Dfs(board, next, str, x + 1, y);  Dfs(board, next, str, x - 1, y);  Dfs(board, next, str, x, y + 1);  Dfs(board, next, str, x, y - 1);  board[y][x] = c;  }  bool IsValidPos(int x, int y)  {  return (x >= 0) && (x < m\_) && (y >= 0) && (y < n\_);  }  private:  set<string> findWords\_;  int m\_{0};  int n\_{0};  int max\_len{ 0 };  }; |

**421. Maximum XOR of Two Numbers in an Array**

给定一个非空数组，数组中元素为 a0, a1, a2, … , an-1，其中 0 ≤ ai < 231 。

找到 ai 和aj 最大的异或 (XOR) 运算结果，其中0 ≤ i,  j < n 。

你能在O(n)的时间解决这个问题吗？

输入: [3, 10, 5, 25, 2, 8]

输出: 28

解释: 最大的结果是 5 ^ 25 = 28.

1、二进制下，我们希望一个数尽可能大，即希望越高位上越能够出现“1”，这样这个数就是所求的最大数，这是贪心算法的思想。

2、于是，我们可以从最高位开始，到最低位，首先假设高位是 “1”，把这 n 个数全部遍历一遍，看看这一位是不是真的可以是“1”，否则这一位就得是“0”

|  |
| --- |
| class Solution {  public:  int findMaximumXOR(vector<int>& nums) {  int N = nums.size();  int res = 0;  int mask = 0;  for(int j = 31; j >= 0; j--) {  mask = mask | (1 << j);  unordered\_set<int> dp;  int temp = res | (1 << j);  for(int i = 0; i < N; i++) {  if (dp.find((nums[i] & mask)^temp) != dp.end()) {  res = temp;  break;  } else {  // 其实由于2个数不轮什么顺序都可以，所以没有必要提前全部存好  dp.insert(nums[i] & mask);  }  }  }  return res;  }  }; |

|  |
| --- |
| class Solution {  public:  struct Trie{  struct TrieNode  {  bool end;  TrieNode\* child[2];  };  TrieNode \* root;  Trie(){  root = new TrieNode();  }  void insert(int x)  {  TrieNode \*p = root;  for(int i = 31; i >=0; --i)  {  int bit = (x >> i) & 1;  if(!p->child[bit])  p->child[bit] = new TrieNode();  p = p->child[bit];  }  p->end = true;  }  int search\_max\_XOR(int x)  {  TrieNode \*p = root;  int ans = 0;  for(int i = 31; i >= 0; --i)  {  int bit = (x >> i) & 1;  if(p->child[bit^1])  {  p = p->child[bit^1];  ans = ans | (1 << i);  }  else  p = p->child[bit];  }  return ans;  }  };  int findMaximumXOR(vector<int>& nums) {  int n = nums.size();  if(n == 0 || n == 1) return 0;  Trie t;  t.insert(nums[0]);  int ans = INT\_MIN;  for(int i = 1; i < n; ++i)  {  ans = max(ans, t.search\_max\_XOR(nums[i]));  t.insert(nums[i]);  }  return ans;  }  }; |

**472. 连接词**

给定一个不含重复单词的列表，编写一个程序，返回给定单词列表中所有的连接词。

连接词的定义为：一个字符串完全是由至少两个给定数组中的单词组成的。

输入: ["cat","cats","catsdogcats","dog","dogcatsdog","hippopotamuses","rat","ratcatdogcat"]

输出: ["catsdogcats","dogcatsdog","ratcatdogcat"]

解释: "catsdogcats"由"cats", "dog" 和 "cats"组成;

"dogcatsdog"由"dog", "cats"和"dog"组成;

"ratcatdogcat"由"rat", "cat", "dog"和"cat"组成。

|  |
| --- |
| class Solution {  public:  vector<string> findAllConcatenatedWordsInADict(vector<string>& words) {  if (words.size() <= 2)  return {};  unordered\_set<string> s(words.begin(), words.end());  vector<string> res;  for (auto word : words) {  if (word.empty()) {  continue;  }  if (dfs(word, 0, 0, s)) {  res.push\_back(word);  }  }  return res;  }  bool dfs(string& word, int start, int cnt, unordered\_set<string>& s) {  if (start >= word.size()) {  return cnt >= 2;  }  for (int i = start; i < word.size(); ++i) {  string t = word.substr(start, i - start + 1);  if (s.count(t) && dfs(word, i+1, cnt+1, s)) {  return true;  }  }  return false;  }  };  class Solution {  public:  vector<string> findAllConcatenatedWordsInADict(vector<string>& words) {  TrieNode trie;  for (const auto& word : words) {  trie.Insert(word);  }  vector<string> res;  for (const auto& word : words) {  int counts = 0;  if (trie.Search(word, 0, 0)) {  res.push\_back(word);  }  }  return res;  }  }; |

|  |
| --- |
| class TrieNode {  public:  TrieNode() {  linkNode\_.resize(MAX\_LINK\_NODE);  }  ~TrieNode() {  for (auto ptr : linkNode\_) {  if (ptr != nullptr) {  delete ptr;  }  }  }  void Insert(const string& word) {  TrieNode\* cur = this;  for (char c : word) {  if (cur->linkNode\_[c - 'a'] == nullptr) {  cur->linkNode\_[c - 'a'] = new TrieNode();  }  cur = cur->linkNode\_[c - 'a'];  }  cur->isEnd\_ = true;  }  bool Search(const string& word) {  // printf("Search: %s\n", word.c\_str());  TrieNode\* cur = this;  for (char c : word) {  if (cur->linkNode\_[c - 'a'] == nullptr) {  return false;  }  cur = cur->linkNode\_[c - 'a'];  }  return cur->isEnd\_;  }  bool Search(const string& word, int index, int counts) {  // printf("Search: %s [%d]=%c \n", word.c\_str(), index, word[index]);  TrieNode\* cur = this;  for (int i = index; i < word.size(); ++i) {  char c = word[i];  if (cur->linkNode\_[c - 'a'] == nullptr) {  return false;  }  cur = cur->linkNode\_[c - 'a'];  if (cur->isEnd\_) {  if (i == word.size() - 1) {  // printf("isEnd: %s [%d]=%c \n", word.c\_str(), i, c);  return counts >= 1;  }  if (Search(word, i + 1, counts + 1)) {  return true;  }  }  }  return false;  }  bool StartWith(const string& prefix) {  // printf("StartWith: %s\n", prefix.c\_str());  TrieNode\* cur = this;  for (char c : prefix) {  if (cur->linkNode\_[c - 'a'] == nullptr) {  return false;  }  cur = cur->linkNode\_[c - 'a'];  }  return true;  }  private:  static const int MAX\_LINK\_NODE{ 26 };  vector<TrieNode\*> linkNode\_;  bool isEnd\_{false};  };  class Solution {  public:  vector<string> findAllConcatenatedWordsInADict(vector<string>& words) {  TrieNode trie;  for (const auto& word : words) {  trie.Insert(word);  }  vector<string> res;  for (const auto& word : words) {  int counts = 0;  if (trie.Search(word, 0, 0)) {  res.push\_back(word);  }  }  return res;  }  }; |

**面试题 17.17. 多次搜索**

给定一个较长字符串big和一个包含较短字符串的数组smalls，设计一个方法，根据smalls中的每一个较短字符串，对big进行搜索。输出smalls中的字符串在big里出现的所有位置positions，其中positions[i]为smalls[i]出现的所有位置。

输入：big = "mississippi"

smalls = ["is","ppi","hi","sis","i","ssippi"]

输出： [[1,4],[8],[],[3],[1,4,7,10],[5]]

|  |
| --- |
| class Solution {  public:  vector<vector<int>> multiSearch(string big, vector<string>& smalls) {  map<char,set<int>>t;  for(int i=0;i<big.size();i++){  t[big[i]].insert(i);  }  vector<vector<int>>res;  for(auto word:smalls){  vector<int>tmp;  char c=word[0];  for(auto pos:t[c]){  if(big.substr(pos,word.size())==word){  tmp.push\_back(pos);  }  }  res.push\_back(tmp);  }  return res;  }  }; |

|  |
| --- |
| struct TrieNode{  int sid;  TrieNode \*child[26];  TrieNode(){  sid=-1;  for(int i=0;i<26;++i) child[i]=NULL;  }  };  class Solution {  private:  TrieNode \*root=new TrieNode();  public:  void insert(string word,int s){  int n=word.size();  TrieNode \*cur=root;  for(int i=0;i<n;++i){  int cid=word.at(i)-'a';  if(cur->child[cid]==NULL) cur->child[cid]=new TrieNode();  cur=cur->child[cid];  }  cur->sid=s;  }  void search(string word,vector<vector<int>>& ans,int bid){  int n=word.size();  TrieNode \*cur=root;  for(int i=0;i<n;++i){  int cid=word.at(i)-'a';  if(cur->sid!=-1) ans[cur->sid].push\_back(bid);  if(cur->child[cid]==NULL) return ;  cur=cur->child[cid];  }  if(cur->sid!=-1) ans[cur->sid].push\_back(bid);  }  vector<vector<int>> multiSearch(string big, vector<string>& smalls) {  int n=smalls.size(),m=big.size();  vector<vector<int>> ans(n,vector<int>{});  for(int i=0;i<n;++i){  if(smalls[i].size()==0) continue;  insert(smalls[i],i);  }  for(int i=0;i<m;++i){  string word=big.substr(i,m-i);  search(word,ans,i);  }  return ans;  }  }; |