宫水三叶的刷题日记

字典树

Author ：宫水三叶
Date ：2021/10/07
QQ Group: 703311589
WeChat ：oaoaya

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

噔噔噔噔，这是公众号「宫水三叶的刷题日记」的原创专题「字典树」合集。

本合集更新时间为 2021-10-07，大概每 2-4 周会集中更新一次。关注公众号，后台回复「字典树」即可获取最新下载链接。

💡下面介绍使用本合集的最佳使用实践：

学习算法：

1. 打开在线目录（Github 版 & Gitee 版）；
2. 从侧边栏的类别目录找到「字典树」；
3. 按照「推荐指数」从大到小进行刷题，「推荐指数」相同，则按照「难度」从易到难进行刷题‘
4. 拿到题号之后，回到本合集进行检索。

维持熟练度：

1. 按照本合集「从上往下」进行刷题。

学习过程中遇到任何困难，欢迎加入「每日一题打卡 QQ 群：703311589」进行交流 🍭🍭🍭

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

这是 LeetCode 上的 **208. 实现 Trie (前缀树)** ，难度为 **中等**。

Tag：「Trie」、「字典树」

Trie（发音类似 “try”）或者说 前缀树 是一种树形数据结构，用于高效地存储和检索字符串数据集中的键。这一数据结构有相当多的应用情景，例如自动补完和拼写检查。

请你实现 Trie 类：

- Trie() 初始化前缀树对象。
- void insert(String word) 向前缀树中插入字符串 word 。
- boolean search(String word) 如果字符串 word 在前缀树中，返回 true（即，在检索之前已经插入）；否则，返回 false 。

- boolean startsWith(String prefix) 如果之前已经插入的字符串 word 的前缀之一为 prefix ，返回 true ；否则，返回 false 。

示例：

```
输入
["Trie", "insert", "search", "search", "startsWith", "insert", "search"]
[[], ["apple"], ["apple"], ["app"], ["app"], ["app"], ["app"]]

输出
[null, null, true, false, true, null, true]

解释
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
```

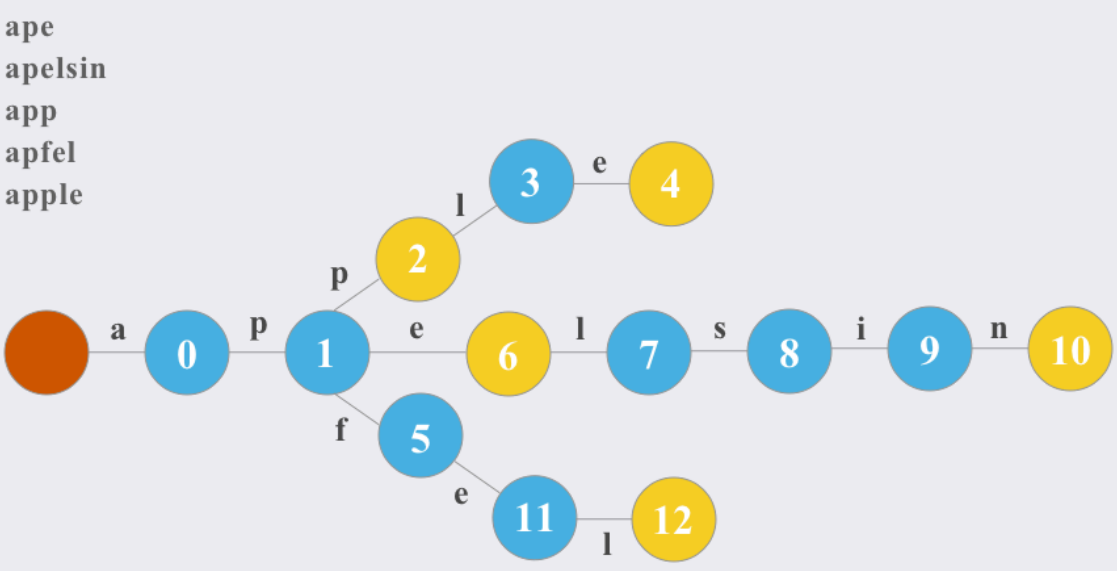
提示：

- 1 <= word.length, prefix.length <= 2000
- word 和 prefix 仅由小写英文字母组成
- insert、search 和 startsWith 调用次数 总计 不超过 $3 * 10^4$ 次

Trie 树

$Trie$ 树（又叫「前缀树」或「字典树」）是一种用于快速查询「某个字符串/字符前缀」是否存在的数据结构。

其核心是使用「边」来代表有无字符，使用「点」来记录是否为「单词结尾」以及「其后续字符串的字符是什么」。



二维数组

一个朴素的想法是直接使用「二维数组」来实现 $Trie$ 树。

- 使用二维数组 $trie[]$ 来存储我们所有的单词字符。
- 使用 $index$ 来自增记录我们到底用了多少个格子（相当于给被用到格子进行编号）。
- 使用 $count[]$ 数组记录某个格子被「被标记为结尾的次数」（当 idx 编号的格子被标记了 n 次，则有 $cnt[idx] = n$）。

代码 ：

```Java
class Trie {
    int N = 100009; // 直接设置为十万级
    int[][] trie;
    int[] count;
    int index;

    public Trie() {
        trie = new int[N][26];
        count = new int[N];
        index = 0;
    }

    public void insert(String s) {
        int p = 0;
        for (int i = 0; i < s.length(); i++) {
            int u = s.charAt(i) - 'a';
            if (trie[p][u] == 0) trie[p][u] = ++index;
            p = trie[p][u];
        }
        count[p]++;
    }

    public boolean search(String s) {
        int p = 0;
        for (int i = 0; i < s.length(); i++) {
            int u = s.charAt(i) - 'a';
            if (trie[p][u] == 0) return false;
            p = trie[p][u];
        }
        return count[p] != 0;
    }

    public boolean startsWith(String s) {
        int p = 0;
        for (int i = 0; i < s.length(); i++) {
            int u = s.charAt(i) - 'a';
            if (trie[p][u] == 0) return false;
            p = trie[p][u];
        }
        return true;
    }
}
```

- 时间复杂度：$Trie$ 树的每次调用时间复杂度取决于入参字符串的长度。复杂度为 $O(Len)$。

- 空间复杂度：二维数组的高度为 n，字符集大小为 k。复杂度为 $O(nk)$。

TrieNode

相比二维数组，更加常规的做法是建立 $TrieNode$ 结构节点。

随着数据的不断插入，根据需要不断创建 $TrieNode$ 节点。

代码：

```Java
class Trie {
    class TrieNode {
        boolean end;
        TrieNode[] tns = new TrieNode[26];
    }

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

    public void insert(String s) {
        TrieNode p = root;
        for(int i = 0; i < s.length(); i++) {
            int u = s.charAt(i) - 'a';
            if (p.tns[u] == null) p.tns[u] = new TrieNode();
            p = p.tns[u];
        }
        p.end = true;
    }

    public boolean search(String s) {
        TrieNode p = root;
        for(int i = 0; i < s.length(); i++) {
            int u = s.charAt(i) - 'a';
            if (p.tns[u] == null) return false;
            p = p.tns[u];
        }
        return p.end;
    }

    public boolean startsWith(String s) {
        TrieNode p = root;
        for(int i = 0; i < s.length(); i++) {
            int u = s.charAt(i) - 'a';
            if (p.tns[u] == null) return false;
            p = p.tns[u];
        }
        return true;
    }
}
```

- 时间复杂度：$Trie$ 树的每次调用时间复杂度取决于入参字符串的长度。复杂度为 $O(Len)$。
- 空间复杂度：结点数量为 n，字符集大小为 k。复杂度为 $O(nk)$。

两种方式的对比

使用「二维数组」的好处是写起来飞快，同时没有频繁 new 对象的开销。但是需要根据数据结构范围估算我们的「二维数组」应该开多少行。

坏处是使用的空间通常是「$TrieNode$」方式的数倍，而且由于通常对行的估算会很大，导致使用的二维数组开得很大，如果这时候每次创建 $Trie$ 对象时都去创建数组的话，会比较慢，而且当样例多的时候甚至会触发 GC（因为 OJ 每测试一个样例会创建一个 $Trie$ 对象）。

因此还有一个小技巧是将使用到的数组转为静态，然后利用 $index$ 自增的特性在初始化 $Trie$ 时执行清理工作 & 重置逻辑。

这样的做法能够使评测时间降低一半，运气好的话可以得到一个与「$TrieNode$」方式差不多的时间。

```java
class Trie {
    // 以下 static 成员独一份，被创建的多个 Trie 共用
    static int N = 100009; // 直接设置为十万级
    static int[][] trie = new int[N][26];
    static int[] count = new int[N];
    static int index = 0;

    // 在构造方法中完成重置 static 成员数组的操作
    // 这样做的目的是为减少 new 操作（无论有多少测试数据，上述 static 成员只会被 new 一次）
    public Trie() {
        for (int row = index; row >= 0; row--) {
            Arrays.fill(trie[row], 0);
        }
        Arrays.fill(count, 0);
        index = 0;
    }

    public void insert(String s) {
        int p = 0;
        for (int i = 0; i < s.length(); i++) {
            int u = s.charAt(i) - 'a';
            if (trie[p][u] == 0) trie[p][u] = ++index;
            p = trie[p][u];
        }
        count[p]++;
    }

    public boolean search(String s) {
        int p = 0;
        for (int i = 0; i < s.length(); i++) {
            int u = s.charAt(i) - 'a';
            if (trie[p][u] == 0) return false;
            p = trie[p][u];
        }
        return count[p] != 0;
    }

    public boolean startsWith(String s) {
        int p = 0;
        for (int i = 0; i < s.length(); i++) {
            int u = s.charAt(i) - 'a';
            if (trie[p][u] == 0) return false;
            p = trie[p][u];
        }
        return true;
    }
```

```
}
```

关于「二维数组」是如何工作 & 1e5 大小的估算

要搞懂为什么行数估算是 1e5，首先要搞清楚「二维数组」是如何工作的。

在「二维数组」中，我们是通过 $index$ 自增来控制使用了多少行的。

当我们有一个新的字符需要记录，我们会将 $index$ 自增（代表用到了新的一行），然后将这新行的下标记录到当前某个前缀的格子中。

举个🌰，假设我们先插入字符串 `abc` 这时候，前面三行会被占掉。

- 第 0 行 `a` 所对应的下标有值，值为 1，代表前缀 `a` 后面接的字符串会被记录在下标为 1 的行内
- 第 1 行 `b` 所对应的下标有值，值为 2，代表前缀 `ab` 后面接的字符串会被记录在下标为 2 的行内
- 第 2 行 `c` 所对应的下标有值，值为 3，代表前缀 `abc` 后面接的字符串会被记录在下标为 3 的行内

当再插入 `abcl` 的时候，这时候会先定位到 `abl` 的前缀行（第 3 行），将 `l` 的下标更新为 4，代表 `abcl` 被加入前缀树，并且前缀 `abcl` 接下来会用到第 4 行进行记录。

但当插入 `abl` 的时候，则会定位到 `ab` 的前缀行（第 2 行），然后将 `l` 的下标更新为 5，代表 `abl` 被加入前缀树，并且前缀 `abl` 接下来会使用第 5 行进行记录。

当搞清楚了「二维数组」是如何工作之后，我们就能开始估算会用到多少行了，调用次数为 10^4，传入的字符串长度为 10^3，假设每一次的调用都是 $insert$，并且每一次调用都会使用到新的 10^3 行。那么我们的行数需要开到 10^7。

但由于我们的字符集大小只有 26，因此不太可能在 10^4 次调用中都用到新的 10^3 行。

而且正常的测试数据应该是 $search$ 和 $startsWith$ 调用次数大于 $insert$ 才有意义的，一个只有 $insert$ 调用的测试数据，任何实现方案都能 AC。

因此我设定了 10^5 为行数估算，当然直接开到 10^6 也没有问题。

关于 Trie 的应用面

首先，在纯算法领域，前缀树算是一种较为常用的数据结构。

不过如果在工程中，不考虑前缀匹配的话，基本上使用 hash 就能满足。

如果考虑前缀匹配的话，工程也不会使用 Trie 。

一方面是字符集大小不好确定（题目只考虑 26 个字母，字符集大小限制在较小的 26 内）因此可以使用 Trie，但是工程一般兼容各种字符集，一旦字符集大小很大的话，Trie 将会带来很大的空间浪费。

另外，对于个别的超长字符 Trie 会进一步变深。

这时候如果 Trie 是存储在硬盘中，Trie 结构过深带来的影响是多次随机 IO，随机 IO 是成本很高的操作。

同时 Trie 的特殊结构，也会为分布式存储将会带来困难。

因此在工程领域中 Trie 的应用面不广。

至于一些诸如「联想输入」、「模糊匹配」、「全文检索」的典型场景在工程主要是通过 ES (ElasticSearch) 解决的。

而 ES 的实现则主要是依靠「倒排索引」

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

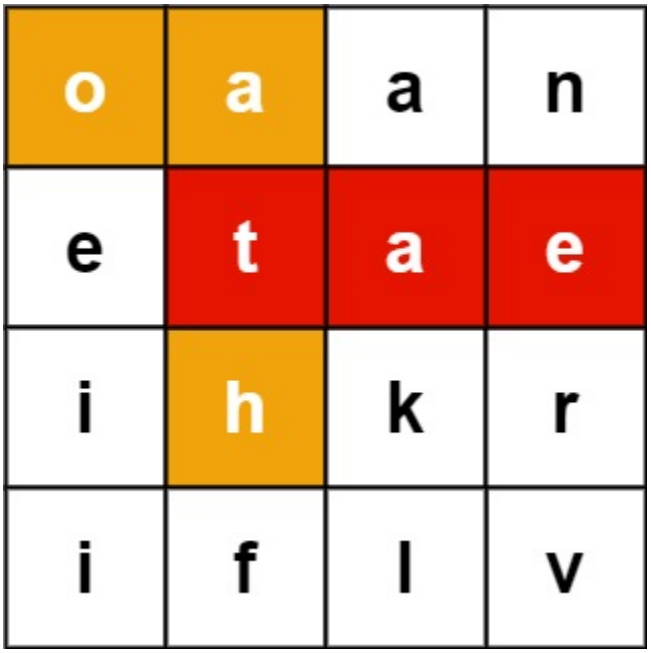
这是 LeetCode 上的 **212. 单词搜索 II** ，难度为 **困难**。

Tag：「回溯算法」、「DFS」、「字典树」

给定一个 m x n 二维字符网格 board 和一个单词（字符串）列表 words，找出所有同时在二维网格和字典中出现的单词。

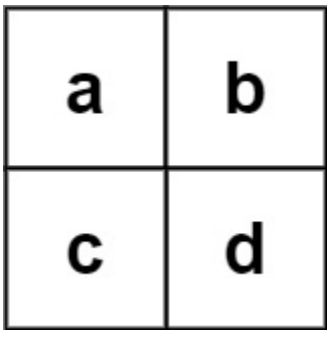
单词必须按照字母顺序，通过 相邻的单元格 内的字母构成，其中“相邻”单元格是那些水平相邻或垂直相邻的单元格。同一个单元格内的字母在一个单词中不允许被重复使用。

示例 1：




```
输入：board = [["o","a","a","n"],["e","t","a","e"],["i","h","k","r"],["i","f","l","v"]], words = ["oath","p
输出：["eat","oath"]
```

示例 2：




```
输入：board = [["a","b"],["c","d"]], words = ["abcb"]
输出：[]
```


提示：

- m == board.length

- n == board[i].length
- 1 <= m, n <= 12
- board[i][j] 是一个小写英文字母
- 1 <= words.length <= 3 * 10^4
- 1 <= words[i].length <= 10
- words[i] 由小写英文字母组成
- words 中的所有字符串互不相同

回溯算法

数据范围只有 12，且 `words` 中出现的单词长度不会超过 10，可以考虑使用「回溯算法」。

起始先将所有 `words` 出现的单词放到 `Set` 结构中，然后以 `board` 中的每个点作为起点进行爆搜（由于题目规定在一个单词中每个格子只能被使用一次，因此还需要一个 `vis` 数组来记录访问过的位置）：

1. 如果当前爆搜到的字符串长度超过 10，直接剪枝；
2. 如果当前搜索到的字符串在 `Set` 中，则添加到答案（同时了防止下一次再搜索到该字符串，需要将该字符串从 `Set` 中移除）。

代码：

```Java
class Solution {
    Set<String> set = new HashSet<>();
    List<String> ans = new ArrayList<>();
    char[][] board;
    int[][] dirs = new int[][]{{1,0},{-1,0},{0,1},{0,-1}};
    int n, m;
    boolean[][] vis = new boolean[15][15];
    public List<String> findWords(char[][] _board, String[] words) {
        board = _board;
        m = board.length; n = board[0].length;
        for (String w : words) set.add(w);
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                vis[i][j] = true;
                sb.append(board[i][j]);
                dfs(i, j, sb);
                vis[i][j] = false;
                sb.deleteCharAt(sb.length() - 1);
            }
        }
        return ans;
    }
    void dfs(int i, int j, StringBuilder sb) {
        if (sb.length() > 10) return ;
        if (set.contains(sb.toString())) {
            ans.add(sb.toString());
            set.remove(sb.toString());
        }
        for (int[] d : dirs) {
            int dx = i + d[0], dy = j + d[1];
            if (dx < 0 || dx >= m || dy < 0 || dy >= n) continue;
            if (vis[dx][dy]) continue;
            vis[dx][dy] = true;
            sb.append(board[dx][dy]);
            dfs(dx, dy, sb);
            vis[dx][dy] = false;
            sb.deleteCharAt(sb.length() - 1);
        }
    }
}
```

- 时间复杂度：共有 $m * n$ 个起点，每次能往 4 个方向搜索（不考虑重复搜索问题），且搜索的长度不会超过 10。整体复杂度为 $O(m * n * 4^{10})$
- 空间复杂度：$O(\sum_{i=0}^{words.length-1} words[i].length)$

Trie

在「解法一」中，对于任意一个当前位置 (i, j)，我们都不可避免的搜索了四联通的全部方向，这导致了那些无效搜索路径最终只有长度达到 10 才会被剪枝。

要进一步优化我们的搜索过程，需要考虑如何在每一步的搜索中进行剪枝。

我们可以使用 $Trie$ 结构进行建树，对于任意一个当前位置 (i, j) 而言，只有在 $Trie$ 中存在往从字符 a 到 b 的边时，我们才在棋盘上搜索从 a 到 b 的相邻路径。

不了解 $Trie$ 的同学，可以看看这篇题解 (题解) 208. 实现 Trie (前缀树)，里面写了两种实现 $Trie$ 的方式。

对于本题，我们可以使用「TrieNode」的方式进行建 $Trie$。

因为 `words` 里最多有 10^4 个单词，每个单词长度最多为 10，如果开成静态数组的话，不考虑共用行的问题，我们需要开一个大小为 $10^5 * 26$ 的大数组，可能会有 TLE 或 MLE 的风险。

与此同时，我们需要将平时建 $TrieNode$ 中的 `isEnd` 标记属性直接换成记录当前字符 `s`，这样我们在 `DFS` 的过程中则无须额外记录当前搜索字符串。

代码：

```java
class Solution {
    class TrieNode {
        String s;
        TrieNode[] tns = new TrieNode[26];
    }
    void insert(String s) {
        TrieNode p = root;
        for (int i = 0; i < s.length(); i++) {
            int u = s.charAt(i) - 'a';
            if (p.tns[u] == null) p.tns[u] = new TrieNode();
            p = p.tns[u];
        }
        p.s = s;
    }
    Set<String> set = new HashSet<>();
    char[][] board;
    int n, m;
    TrieNode root = new TrieNode();
    int[][] dirs = new int[][]{{1,0},{-1,0},{0,1},{0,-1}};
    boolean[][] vis = new boolean[15][15];
    public List<String> findWords(char[][] _board, String[] words) {
        board = _board;
        m = board.length; n = board[0].length;
        for (String w : words) insert(w);
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                int u = board[i][j] - 'a';
                if (root.tns[u] != null) {
                    vis[i][j] = true;
                    dfs(i, j, root.tns[u]);
                    vis[i][j] = false;
                }
            }
        }
        List<String> ans = new ArrayList<>();
        for (String s : set) ans.add(s);
        return ans;
    }
    void dfs(int i, int j, TrieNode node) {
        if (node.s != null) set.add(node.s);
        for (int[] d : dirs) {
            int dx = i + d[0], dy = j + d[1];
            if (dx < 0 || dx >= m || dy < 0 || dy >= n) continue;
            if (vis[dx][dy]) continue;
            int u = board[dx][dy] - 'a';
            if (node.tns[u] != null) {
```

```
                vis[dx][dy] = true;
                dfs(dx, dy, node.tns[u]);
                vis[dx][dy] = false;
            }
        }
    }
}
```

- 时间复杂度：共有 $m * n$ 个起点，每次能往 4 个方向搜索（不考虑重复搜索问题），且搜索的长度不会超过 10。整体复杂度为 $O(m * n * 4^{10})$
- 空间复杂度：$O(\sum_{i=0}^{words.length-1} words[i].length * C)$，$C$ 为字符集大小，固定为 26

题目描述

这是 LeetCode 上的 **1707. 与数组中元素的最大异或值** ，难度为 **困难**。

Tag：「Trie」、「字典树」、「二分」

给你一个由非负整数组成的数组 nums 。另有一个查询数组 queries ，其中 queries[i] = [xi, mi] 。

第 i 个查询的答案是 xi 和任何 nums 数组中不超过 mi 的元素按位异或（XOR）得到的最大值。

换句话说，答案是 max(nums[j] XOR xi) ，其中所有 j 均满足 nums[j] <= mi 。如果 nums 中的所有元素都大于 mi，最终答案就是 -1 。

返回一个整数数组 answer 作为查询的答案，其中 answer.length == queries.length 且 answer[i] 是第 i 个查询的答案。

示例 1：

```
输入：nums = [0,1,2,3,4], queries = [[3,1],[1,3],[5,6]]

输出：[3,3,7]

解释：
1) 0 和 1 是仅有的两个不超过 1 的整数。0 XOR 3 = 3 而 1 XOR 3 = 2 。二者中的更大值是 3 。
2) 1 XOR 2 = 3.
3) 5 XOR 2 = 7.
```

示例 2：

```
输入：nums = [5,2,4,6,6,3], queries = [[12,4],[8,1],[6,3]]

输出：[15,-1,5]
```

提示：

- 1 <= nums.length, queries.length <= 10^5
- queries[i].length == 2
- 0 <= nums[j], xi, mi <= 10^9

基本分析

在做本题之前，请先确保已经完成 421. 数组中两个数的最大异或值。

这种提前给定了所有询问的题目，我们可以运用离线思想（调整询问的回答顺序）进行求解。

对于本题有两种离线方式可以进行求解。

普通 Trie

第一种方法基本思路是：**不一次性地放入所有数，而是每次将需要参与筛选的数字放入 $Trie$，再进行与 421. 数组中两个数的最大异或值 类似的贪心查找逻辑。**

具体的，我们可以按照下面的逻辑进行处理：

1. 对 `nums` 进行「从小到大」进行排序，对 `queries` 的第二维进行「从小到大」排序（排序前先将询问原本的下标映射关系存下来）。
2. 按照排序顺序处理所有的 `queries[i]`：
    1. 在回答每个询问前，将小于等于 `queries[i][1]` 的数值存入 $Trie$。由于我们已经事先对 `nums` 进行排序，因此这个过程只需要维护一个在 `nums` 上有往右移动的指针即可。
    2. 然后利用贪心思路，查询每个 `queries[i][0]` 所能找到的最大值是多少，计算异或和（此过程与 421. 数组中两个数的最大异或值 一致）。
    3. 找到当前询问在原询问序列的下标，将答案存入。

代码：

```java
class Solution {
    static int N = (int)1e5 * 32;
    static int[][] trie = new int[N][2];
    static int idx = 0;
    public Solution() {
        for (int i = 0; i <= idx; i++) {
            Arrays.fill(trie[i], 0);
        }
        idx = 0;
    }
    void add(int x) {
        int p = 0;
        for (int i = 31; i >= 0; i--) {
            int u = (x >> i) & 1;
            if (trie[p][u] == 0) trie[p][u] = ++idx;
            p = trie[p][u];
        }
    }
    int getVal(int x) {
        int ans = 0;
        int p = 0;
        for (int i = 31; i >= 0; i--) {
            int a = (x >> i) & 1, b = 1 - a;
            if (trie[p][b] != 0) {
                p = trie[p][b];
                ans = ans | (b << i);
            } else {
                p = trie[p][a];
                ans = ans | (a << i);
            }
        }
        return ans ^ x;
    }
    public int[] maximizeXor(int[] nums, int[][] qs) {
        int m = nums.length, n = qs.length;

        // 使用哈希表将原本的顺序保存下来
        Map<int[], Integer> map = new HashMap<>();
        for (int i = 0; i < n; i++) map.put(qs[i], i);

        // 将 nums 与 queries[x][1] 进行「从小到大」进行排序
        Arrays.sort(nums);
        Arrays.sort(qs, (a, b)->a[1]-b[1]);

        int[] ans = new int[n];
        int loc = 0; // 记录 nums 中哪些位置之前的数已经放入 Trie
```

```
        for (int[] q : qs) {
            int x = q[0], limit = q[1];
            // 将小于等于 limit 的数存入 Trie
            while (loc < m && nums[loc] <= limit) add(nums[loc++]);
            if (loc == 0) {
                ans[map.get(q)] = -1;
            } else {
                ans[map.get(q)] = getVal(x);
            }
        }
        return ans;
    }
}
```

- 时间复杂度：令 `nums` 的长度为 `m`，`qs` 的长度为 `n`。两者排序的复杂度为 $O(m \log m)$ 和 $O(n \log n)$；将所有数插入 $Trie$ 和从 $Trie$ 中查找的复杂度均为 $O(Len)$，Len 为 32。
  整体复杂度为 $O(m \log m + n \log n + (m + n) * Len) = O(m * \max(\log m, Len) + n * \max(\log n, Len))$。
- 空间复杂度：$O(C)$。其中 C 为常数，固定为 $1e5 * 32 * 2$。

计数 Trie & 二分

另外一个比较「增加难度」的做法是，将整个过程翻转过来：**一次性存入所有的 $Trie$ 中，然后每次将不再参与的数从 $Trie$ 中移除。相比于解法一，这就要求我们为 $Trie$ 增加一个「删除/计数」功能，并且需要实现二分来找到移除元素的上界下标是多少。**

具体的，我们可以按照下面的逻辑进行处理：

1. 对 `nums` 进行「从大到小」进行排序，对 `queries` 的第二维进行「从大到小」排序（排序前先将询问原本的下标映射关系存下来）。
2. 按照排序顺序处理所有的 `queries[i]`：
    1. 在回答每个询问前，通过「二分」找到在 `nums` 中第一个满足「小于等于 `queries[i][1]` 的下标在哪」，然后将该下标之前的数从 $Trie$ 中移除。同理，这个过程我们需要使用一个指针来记录上一次删除的下标位置，避免重复删除。
    2. 然后利用贪心思路，查询每个 `queries[i][0]` 所能找到的最大值是多

少。注意这是要判断当前节点是否有被计数，如果没有则返回 -1。

3. 找到当前询问在原询问序列的下标，将答案存入。

代码：

```java
class Solution {
    static int N = (int)1e5 * 32;
    static int[][] trie = new int[N][2];
    static int[] cnt = new int[N];
    static int idx = 0;
    public Solution() {
        for (int i = 0; i <= idx; i++) {
            Arrays.fill(trie[i], 0);
            cnt[i] = 0;
        }
        idx = 0;
    }
    // 往 Trie 存入(v = 1)/删除(v = -1) 某个数 x
    void add(int x, int v) {
        int p = 0;
        for (int i = 31; i >= 0; i--) {
            int u = (x >> i) & 1;
            if (trie[p][u] == 0) trie[p][u] = ++idx;
            p = trie[p][u];
            cnt[p] += v;
        }
    }
    int getVal(int x) {
        int ans = 0;
        int p = 0;
        for (int i = 31; i >= 0; i--) {
            int a = (x >> i) & 1, b = 1 - a;
            if (cnt[trie[p][b]] != 0) {
                p = trie[p][b];
                ans = ans | (b << i);
            } else if (cnt[trie[p][a]] != 0) {
                p = trie[p][a];
                ans = ans | (a << i);
            } else {
                return -1;
            }
        }
        return ans ^ x;
    }
    public int[] maximizeXor(int[] nums, int[][] qs) {
        int n = qs.length;

        // 使用哈希表将原本的顺序保存下来
        Map<int[], Integer> map = new HashMap<>();
        for (int i = 0; i < n; i++) map.put(qs[i], i);
```

```
        // 对两者排降序
        sort(nums);
        Arrays.sort(qs, (a, b)->b[1]-a[1]);

        // 将所有数存入 Trie
        for (int i : nums) add(i, 1);

        int[] ans = new int[n];
        int left = -1; // 在 nums 中下标「小于等于」left 的值都已经从 Trie 中移除
        for (int[] q : qs) {
            int x = q[0], limit = q[1];
            // 二分查找到待删除元素的右边界，将其右边界之前的所有值从 Trie 中移除。
            int right = getRight(nums, limit);
            for (int i = left + 1; i < right; i++) add(nums[i], -1);
            left = right - 1;
            ans[map.get(q)] = getVal(x);
        }
        return ans;
    }
    // 二分找到待删除的右边界
    int getRight(int[] nums, int limit) {
        int l = 0, r = nums.length - 1;
        while (l < r) {
            int mid = l + r >> 1;
            if (nums[mid] <= limit) {
                r = mid;
            } else {
                l = mid + 1;
            }
        }
        return nums[r] <= limit ? r : r + 1;
    }
    // 对 nums 进行降序排序（Java 没有 Api 直接支持对基本类型 int 排倒序，其他语言可忽略）
    void sort(int[] nums) {
        Arrays.sort(nums);
        int l = 0, r = nums.length - 1;
        while (l < r) {
            int c = nums[r];
            nums[r--] = nums[l];
            nums[l++] = c;
        }
    }
}
```

- 时间复杂度：令 `nums` 的长度为 `m`，`qs` 的长度为 `n`，常数 $Len = 32$。两者排序的复杂度为 $O(m \log m)$ 和 $O(n \log n)$；将所有数插入 $Trie$ 的复杂度为

$O(m * Len)$；每个查询都需要经过「二分」找边界，复杂度为 $O(n \log m)$；最坏情况下所有数都会从 $Trie$ 中被标记删除，复杂度为 $O(m * Len)$。
整体复杂度为 $O(m \log m + n \log n + n \log m + mLen) = O(m * \max(\log m, Len) + n * \max(\log m, \log n))$。

- 空间复杂度：$O(C)$。其中 C 为常数，固定为 $1e5 * 32 * 3$。

说明

这两种方法我都是采取「数组实现」，而且由于数据范围较大，都使用了 `static` 来优化大数组创建，具体的「优化原因」与「类实现 Trie 方式」可以在题解 421. 数组中两个数的最大异或值 查看，这里不再赘述。

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

💡**更新 Tips：本专题更新时间为 2021-10-07，大概每 2-4 周 集中更新一次。**

最新专题合集资料下载，可关注公众号「宫水三叶的刷题日记」，回台回复「字典树」获取下载链接。

觉得专题不错，可以请作者吃糖 🍭🍭🍭 ：

"给作者手机充个电"

YOLO 的赞赏码