【第三十周】字典树（Trie）与双数组字典树（Double-Array-Trie）

1、剑指 Offer II 067. 最大的异或

1. 在一个数组中求两个数的异或最大值，因为大的数异或后带来的增益较大，所以优先考虑大的数进行异或，也就是先考虑高位。
2. 当两个二进制位不同，为1；相同为0。所以，我们尽量找到两个很多位都不同的数进行异或。
3. 构建前缀树：将所有的数都插入前缀树中。与普通前缀树不同，这里是根据一个数的二进制位来插入，普通前缀树是根据单词中的字符来插入。还有一点需要注意的是，我们优先考虑高位，所以是从高位到低位进行插入。
4. 对于 nums 中的每个数 num ，都到前缀树中去搜索num最大异或的那个数，然后计算最大异或值，最后，从这些异或值中挑出最大的一个就是要的答案。

```
/**
* @param {number[]} nums
* @return {number}
*/

var findMaximumXOR = function(nums) {
    // 前缀树 先将每一个树的31位二进制位存入到前缀树中
    let root = {};
    for (let num of nums) {
        let cur = root;
        let bit = 0; // 减少空间
        for (let i = 30; i >= 0; i--) { // JS是31位数，所以右移30位
        // 数据是从高位开始存储(后面高位亦或取1，结果肯定是最大的)
            bit = (num >> i) & 1;
            if (!cur[bit]) cur[bit] = {};
            cur = cur[bit];
        }
    }
    // 时间复杂度 O(N) 因为每个数据在前缀树中查找一次就能得到它与其他数xor的最大结果
    // 此处准确时间复杂度应该是 O(31N)
    let ans = 0;
    for (let num of nums) {
        let cur = root;
        let res = 0;
        for (let i = 30; i >= 0; i--) {
            let bit = (num >> i) & 1;
            if (cur[1] && (1 ^ bit == 1)) {
                res = (res << 1) + 1; // 有1取1
                cur = cur[1];
            } else if (cur[0] && (0 ^ bit == 1)) {
                res = (res << 1) + 1; // 有1取1
                cur = cur[0];
            } else { // else 说明不是两个都有的，取一个有的就行
                res = res << 1;
                cur = cur[bit];
            }
```

```
        }
        // 每次计算二进制算一次值
        if (res > ans) ans = res;
    }
    return ans;
};
```

2、1268. 搜索推荐系统

1. 当我们在字典树中插入字符串 product 并遍历到节点 node 时，我们将 product 存储在 node 中，若此时 node 中的字符串超过三个，就丢弃字典序最大的那个字符串。
2. 这样在所有的字符串都被插入到字典树中后，字典树中的节点 node 就存放了当输入为 prefix 时应该返回的那些字符串。

```
// 我们需要将 searchWord 的前缀与 products 中的字符串进行匹配，因此我们可以使用字典树（Trie）来存储 products 中的所有字符串。
// 这样以来，当我们依次输入 searchWord 中的每个字母时，我们可以从字典树的根节点开始向下查找，判断是否存在以当前的输入为前缀的字符串，并找出字典序最小的三个（若存在）字符串。
var suggestedProducts = function(products, searchWord) {
    class Trie{
        constructor(){
            this.child=new Map()
            this.pro=[]
        }
    }
    var addWord=function(trie,str){
        let t=trie
        for(let x of str){
            if(!t.child.has(x)){
                let p=new Trie()
                t.child.set(x,p)
            }
            t=t.child.get(x)
            t.pro.push(str)
            t.pro.sort()
            if(t.pro.length>3){
                t.pro.pop()
            }
        }
    }
    let root = new Trie()
    for(let x of products){
        addWord(root,x)
    }
    let res=[]
    let next=root
    let empty=new Trie()
    for(let x of searchWord){
        if(next.child.has(x)){
            next=next.child.get(x)
            res.push(next.pro)
        }else{
            next=empty
```

```
            res.push([])
        }

    }
    return res
};
```

3、440. 字典序的第K小数字

1. 确定指定前缀下所有子节点数，给定一个前缀，返回下面子节点总数。就是用下一个前缀的起点减去当前前缀的起点，那么就是当前前缀下的所有子节点数总和。
2. 第k个数在当前前缀下，往子树里面去看。prefix *= 10
3. 第k个数不在当前前缀下，当前的前缀小了，我们扩大前缀。

```
/**
 * @param {number} n
 * @param {number} k
 * @return {number}
 */
var findKthNumber = function(n, k) {
  let getCount = (prefix, n) => {
    let count =  0;
    for(let cur = prefix, next = prefix + 1; cur <= n; cur *= 10, next *= 10)
      count += Math.min(next, n+1) - cur;
    return count;
  }
  let p = 1;//作为一个指针，指向当前所在位置，当p==k时，也就是到了排位第k的数
  let prefix = 1;//前缀
  while(p < k) {
    let count = getCount(prefix, n);//获得当前前缀下所有子节点的和
    if(p + count > k) {//第k个数在当前前缀下
      prefix *= 10;
      p++;//把指针指向了第一个子节点的位置，比如11乘10后变成110，指针从11指向了110
    } else if(p + count <= k) {//第k个数不在当前前缀下
      prefix ++;
      p += count;//注意这里的操作，把指针指向了下一前缀的起点
    }
  }
  return prefix;
};
```

4、611. 有效三角形的个数

1. 方法：二分查找
2. 判断三条边能组成三角形的条件为：任意两边之和大于第三边，任意两边之差小于第三边。
3. 三条边长从小到大为 a、b、c，当且仅当 a + b > c 这三条边能组成三角形。
4. 首先对数组排序。固定最短的两条边，二分查找最后一个小于两边之和的位置。
5. 可以求得固定两条边长之和满足条件的结果。枚举结束后，总和就是答案。
6. 时间复杂度为 O(n^2logn)

```
```

```
var triangleNumber = function(nums) {
    const n = nums.length;
    nums.sort((a, b) => a - b);
    let ans = 0;
    for (let i = 0; i < n; ++i) {
        for (let j = i + 1; j < n; ++j) {
            let left = j + 1, right = n - 1, k = j;
            while (left <= right) {
                const mid = Math.floor((left + right) / 2);
                if (nums[mid] < nums[i] + nums[j]) {
                    k = mid;
                    left = mid + 1;
                } else {
                    right = mid - 1;
                }
            }
            ans += k - j;
        }
    }
    return ans;
};
```