【第十六课】月度测试题

同学们注意 这个月我们讲了二分查找，哈希表和布隆过滤器，深搜与广搜。

690. 员工的重要性

方法一：深度优先搜索

1.根据给定的员工编号找到员工，从该员工开始遍历，对于每个员工，将其重要性加到总结中然后对每个直系下属继续遍历，直到所有下属遍历完毕，此时的总和即为给定的员工及其所有下属的重要性之和。

2.实现方面，由于给定是员工的编号，并且每个员工的编号都不相同，因此使用哈希表存储每个员工编号和对应的员工，即可通过编号得到对应的员工。

```
 var GetImportance = function(employees, id) {
    const map = new Map();
    for (const employee of employees) {
        map.set(employee.id, employee);
    }
    const dfs = (id) => {
        const employee = map.get(id);
        let total = employee.importance;
        const subordinates = employee.subordinates;
        for (const subId of subordinates) {
            total += dfs(subId);
        }
        return total;

    }

    return dfs(id);
};
```

方法二：广度优先搜索

1.使用哈希表存储每个员工的员工编号和对应的员工，即可通过员工编号得到对应的员工。

2.根据给定的员工编号找到对应的员工，从该员工进行广度优先搜索，对于每一个被遍历到的员工，将其的重要性加入到总和之中，最终得到的总和就是为给定员工的及其所有下属的重要性之和。

```
var GetImportance = function(employees, id) {
    const map = new Map();
    for (const employee of employees) {
        map.set(employee.id, employee);
    }
    let total = 0;
    const queue = [];
    queue.push(id);
    while (queue.length) {
        const curId = queue.shift();
        const employee = map.get(curId);
        total += employee.importance;
```

```
            const subordinates = employee.subordinates;
            for (const subId of subordinates) {
                queue.push(subId);
            }
        }
        return total;
};
```

17. 电话号码的字母组合

1.方法是广度优先搜索BFS；其实就是将数字串 “翻译”成字母串，找出所有的翻译可能性。

2.翻译第一个数字的可能性是3/4种选择， 翻译第二个数字的可能性也是3/4种选择，.....所以以此类推

3.从首位翻译到最末位，会展开一棵递归树。指针i是当前考察的字符的索引。

4.当指针越界的时候，此时生成一个解，加入解集，结束当前的递归，去别的分支，找齐所有的解。

```
  const letterCombinations = (digits) => {
  if (digits.length == 0) return [];
  const map = { '2': 'abc', '3': 'def', '4': 'ghi', '5': 'jkl', '6': 'mno', '7':
'pqrs', '8': 'tuv', '9': 'wxyz' };

  const queue = [];
  queue.push('');
  for (let i = 0; i < digits.length; i++) { // bfs的层数，即digits的长度
    const levelSize = queue.length;         // 当前层的节点个数
    for (let j = 0; j < levelSize; j++) {   // 逐个让当前层的节点出列
      const curStr = queue.shift();         // 出列

      const letters = map[digits[i]];

      for (const l of letters) {
        queue.push(curStr + l); // 生成新的字母串入列
      }
    }
  }
  return queue; // 队列中全是最后一层生成的字母串
};
```

方法二：深度优先搜索DFS

```
const letterCombinations = (digits) => {
  if (digits.length == 0) return [];
  const res = [];
  const map = { '2': 'abc', '3': 'def', '4': 'ghi', '5': 'jkl', '6': 'mno', '7':
'pqrs', '8': 'tuv', '9': 'wxyz' };
  // dfs: 当前构建的字符串为curStr，现在“翻译”到第i个数字，基于此继续“翻译”
  const dfs = (curStr, i) => {   // curStr是当前字符串，i是扫描的指针
    if (i > digits.length - 1) { // 指针越界，递归的出口
      res.push(curStr);          // 将解推入res
      return;                    // 结束当前递归分支
    }
    const letters = map[digits[i]]; // 当前数字对应的字母
```

```
    for (const letter of letters) { // 一个字母是一个选择，对应一个递归分支
      dfs(curStr + letter, i + 1);  // 选择翻译成letter，生成新字符串，i指针右移继续翻译（递归）
    }
  };
  dfs('', 0); // 递归的入口，初始字符串为''，从下标0开始翻译
  return res;
};
```

279. 完全平方数

1.求出最大平方数，可以得到一个数组【1，4,.....,Ma x】；

2.队列的初始值可以设置为0，也可以设置为n；

3.可以看成N叉树，每次都可以是【1，4，.....，Ma x】种选择进行求和

4.然后采用BFS搜索

5.得到和为n即为返回结果（队列的初始值为n,就是用差判断）

```
var numSquares = function (n) {
    // let sqr = ~~Math.sqrt(n);
    let neighbor = [];
    let queue = new Set([0]) //使用set消除重复，优化效率
    for (let i = 1; i * i <= n; i++) {
        neighbor.push(i * i);//注意这里是从小到大
    }
    let count = 0;
    while (queue.size) {
        let nextSet = new Set();
        count++;
        for (let v of queue) {
            for (let c of neighbor) {
                let add = v + c;
                if (add === n) {
                    return count;
                }
                if (add > n) {
                    //后面都是更大的了
                    break;
                }
                nextSet.add(add);
            }
        }
        queue = nextSet;
    }
    return count;
};
```

111. 二叉树的最小深度

1.做法: DFS终止条件 返回值和递归过程

1.当前节点root 为空时候=，说明此树的高度为0，0也是最小值

2.当前节点root的左子树和右子树都为空的时候，则说明当前节点有值，且需要分别计算其左子树和右子树的最小深度，返回最小深度 +1，+1表示当前节点存在有1个深度。

```
var minDepth = function(root) {
    if(root == null) {
        return 0;
    }
    if(root.left == null && root.right == null) {
        return 1;
    }
    let ans = Number.MAX_SAFE_INTEGER;
    if(root.left != null) {
        ans = Math.min(minDepth(root.left), ans);
    }
    if(root.right != null) {
        ans = Math.min(minDepth(root.right), ans);
    }
    return ans + 1;
};
```

1306. 跳跃游戏 III

题目的内容是一个小游戏，可以想象这样一个场景：

1.面前放着几张纸牌，每个纸牌下面写着一个数字

2.游戏开始时，作为我们起始位置的纸牌已经确定啦

3.把纸牌翻过来，看到后面的数字为 n，那么我们现在可以选择向左走 n 步或者向右走 n 步，但是不能超出纸牌的范围

4.不断的重复这个过程，直到我们遇到翻过来的数字为 0 我们就成功啦

5.如果无论如何都找不到 0，那么我们就失败啦

6.那么回到题目中，纸牌和背后的数字是一个给定的由非负整数组成的数组，起始位置是给定的一个下标，我们需要返回 true 或者 false 。

1.由于出发点是确定的，而结束的点不确定，因为可能会有多个 0 的存在，所以我们可以从出发点开始不断的做尝试。基于此我们可以得到两种思路：

2.遇到了选择左右的情况时，我们把两种情况都记录下来，然后继续针对所有已经记录的内容逐个继续尝试，不过需要注意循环的情况。

3.遇到了选择左右的情况时，先选择一个方向，然后继续走下去，直到发生了循环再退到上一个选择点重新选择。

1.方法一：深度优先搜索

```
const canReach = (arr, start) => {
  const val = arr[start];
  if (val === 0) return true;
  if (val === -1) return false;
  arr[start] = -1;
  return (start - val >= 0 && canReach(arr, start - val)) || (start + val <
arr.length && canReach(arr, start + val));
};
```

方法二：广度优先搜索

```
const canReach = (arr, start) => {
  const visited = new Set();
  const queue = [start];
  for (let len = 0, max = arr.length; len < queue.length; ++len) {
    const idx = queue[len];
    if (visited.has(idx)) continue;
    if (arr[idx] === 0) return true;
    visited.add(idx);
    idx + arr[idx] < max && queue.push(idx + arr[idx]);
    idx - arr[idx] >= 0 && queue.push(idx - arr[idx]);
  }
  return false;
};
```

剑指 Offer 11. 旋转数组的最小数字

1.用二分查找即可

2.若mid大于high的数，则最小值一定在mid右侧

3.若mid小于high的数，则最小值有两种可能：(1)最小值在mid最侧(2)mid就是最小值

4.若mid等于high的数，high--

5.最后返回low所在的数

```
// 二分查找 官方做法
var minArray = function(numbers) {
    let low = 0;
    let high = numbers.length - 1;
    while (low < high) {
        const pivot = low + Math.floor((high - low) / 2);
        if (numbers[pivot] < numbers[high]) {
            high = pivot;
        } else if (numbers[pivot] > numbers[high]) {
            low = pivot + 1;
        } else {
            high -= 1;
        }
    }
    return numbers[low];
};
```

658. 找到 K 个最接近的元素

1. 把mi看成结果的起始下标，判断结果是否正确
2. 如果数组array[mide+k]位置的差值比array[mid]位置的差值要小，那说明起始值比mid大。
3. 因为假设返回值窗口取的值是mid 到 [mid+k],是k+1的长度，不是k的长度。反之起始值肯定是mid或者是在mi d左边。

4. (1)目的找到结果数组的起始下标（2）如果 x - array][mid] > arr[mid+k] - x,那么起始值肯定在mid的右边（3）反之起始值是在mid或者是在mid右边（4）返回起始值到 k - 1个元素

```
var findClosestElements = function(arr, k, x) {
 let low = 0, high = arr.length - 1;
 while (low < high) {
   const mid = low + Math.floor((high - low) / 2);
   x - arr[mid] > arr[mid + k]- x ? low = mid + 1 : high = mid;
 }
 return arr.slice(low, low + k);
};
```

575. 分糖果

1. 数组去重。
2. 如果去重后的长度小于等于原数组长度的一半，返回去重后的长度。
3. 如果去重后的长度大于原数组长度的一半，返回原数组长度的一半。

```
var distributeCandies = function(candyType) {
    let len = candyType.length
    // 拿到 数组中不同的糖果个数
    let n = new Set(candyType).size ;
    // 表示妹妹分得的最大不重复糖果
    let i = n;
    /**
     * len%(2*i)!=0 该假设不成立，该妹妹的糖果全部为不重复
     * (len - 2*i)<0||(len - 2*i)%2!=0) 该条件不成立，妹妹除了分配的糖果为不重复的外，还能与弟弟平均分配重复了的糖果
     * 而如果第一个表达式不成立则第二个必然不成立，第一个成立第二个不一定成立，应为数据是从大到小，所以得到的i为最大不重复糖果数
     */
    while(len%(2*i)!=0&&((len - 2*i)<0||(len - 2*i)%2!=0)) i--;
    return i;
};
```

1487. 保证文件名唯一

做法：哈希。新生成一次名字要再遍历一遍存储的哈希表中是否有相同字符串的key，如果没有，返回的同时记得往哈希表中添加这个新生成的文件名；如果有，就再次生成新的名字再进行判断。

```
var getFolderNames = function(names) {
    let d = {}, ans = []
    for (const name of names) {
        let s = name
        while (s in d) {
            s = name + '(' + d[name] + ')'
            ++d[name]
        }
        d[s] = 1
        ans.push(s)
    }
    return ans
```

```
};
```

310. 最小高度树

方法是dfs

1.建图。

2.遍历图的各个顶点，找出该顶点为根的树的深度。

3.返回最小深度的结果，(细想可知：结果数组只能含 1 或 2个元素)。

从a节点找到最远的b节点，然后b节点开始找到最远的c节点，然后bc间路径的中间一个点或者两个点就是答案；

```
var findMinHeightTrees = function (n, edges) {
    // check
    if (n === 1 || edges.length === 0) return [0];

    let root, len = edges.length, inDegs = new Array(n);
    do {
        // update length of edges
        edges.length = len;
        inDegs.fill(0);
        for (let edge of edges) {
            inDegs[edge[0]]++;
            inDegs[edge[1]]++;
        }

        len = 0;
        for (let edge of edges) {
            // overwrite the value of edges if none of the edge's nodes is leaf
            if (inDegs[edge[0]] > 1 && inDegs[edge[1]] > 1) edges[len++] = edge;
            else if (inDegs[edge[0]] > 1) root = edge[0];
            else if (inDegs[edge[1]] > 1) root = edge[1];
        }
    } while (len) // when len is 0, the edges hold the previous values

    if (edges.length === 1) return edges[0]; // case 1
    return [root]; // case 2
};
```