算法知识总结

本部分主要是笔者在学习算法知识和一些相关面试题所做的笔记，如果出现错误，希望大家指出！

目录

常用算法和数据结构总结

排序

冒泡排序

冒泡排序的基本思想是，对相邻的元素进行两两比较，顺序相反则进行交换，这样，每一趟会将最小或最大的元素“浮”到顶端，
最终达到完全有序。

代码实现：

```
1
2  function bubbleSort(arr) {
3      if (!Array.isArray(arr) || arr.length <= 1) return;
4      let lastIndex = arr.length - 1;
5      while (lastIndex > 0) { // 当最后一个交换的元素为第一个时，说明后面全部排序完毕
6          let flag = true, k = lastIndex;
7          for (let j = 0; j < k; j++) {
8              if (arr[j] > arr[j + 1]) {
9                  flag = false;
10                 lastIndex = j; // 设置最后一次交换元素的位置
11                 [arr[j], arr[j+1]] = [arr[j+1], arr[j]];
12             }
13         }
14         if (flag) break;
15     }
16 }
17
```

冒泡排序有两种优化方式。

一种是外层循环的优化，我们可以记录当前循环中是否发生了交换，如果没有发生交换，则说明该序列已经为有序序列了。
因此我们不需要再执行之后的外层循环，此时可以直接结束。

一种是内层循环的优化，我们可以记录当前循环中最后一次元素交换的位置，该位置以后的序列都是已排好的序列，因此下
一轮循环中无需再去比较。

优化后的冒泡排序，当排序序列为已排序序列时，为最好的时间复杂度为 O(n)。

冒泡排序的平均时间复杂度为 $O(n^2)$，最坏时间复杂度为 $O(n^2)$，空间复杂度为 O(1)，是稳定排序。

详细资料可以参考：
《图解排序算法(一)》
《常见排序算法 - 鸡尾酒排序 》
《前端笔试&面试爬坑系列---算法》
《前端面试之道》

选择排序

选择排序的基本思想为每一趟从待排序的数据元素中选择最小（或最大）的一个元素作为首元素，直到所有元素排完为止。

在算法实现时，每一趟确定最小元素的时候会通过不断地比较交换来使得首位置为当前最小，交换是个比较耗时的操作。其实
我们很容易发现，在还未完全确定当前最小元素之前，这些交换都是无意义的。我们可以通过设置一个变量 min，每一次比较
仅存储较小元素的数组下标，当轮循环结束之后，那这个变量存储的就是当前最小元素的下标，此时再执行交换操作即可。

代码实现：

```
function selectSort(array) {

  let length = array.length;

  // 如果不是数组或者数组长度小于等于1，直接返回，不需要排序
  if (!Array.isArray(array) || length <= 1) return;

  for (let i = 0; i < length - 1; i++) {

    let minIndex = i; // 设置当前循环最小元素索引

    for (let j = i + 1; j < length; j++) {

      // 如果当前元素比最小元素索引，则更新最小元素索引
      if (array[minIndex] > array[j]) {
        minIndex = j;
      }
    }

    // 交换最小元素到当前位置
    // [array[i], array[minIndex]] = [array[minIndex], array[i]];
    swap(array, i, minIndex);
```

```
  }

  return array;
}

// 交换数组中两个元素的位置
function swap(array, left, right) {
  var temp = array[left];
  array[left] = array[right];
  array[right] = temp;
}
```

选择排序不管初始序列是否有序，时间复杂度都为 $O(n^2)$。

选择排序的平均时间复杂度为 $O(n^2)$ ，最坏时间复杂度为 $O(n^2)$ ，空间复杂度为 $O(1)$ ，不是稳定排序。

详细资料可以参考：
《图解排序算法(一)》

插入排序

直接插入排序基本思想是每一步将一个待排序的记录，插入到前面已经排好序的有序序列中去，直到插完所有元素为止。

插入排序核心--扑克牌思想： 就想着自己在打扑克牌，接起来一张，放哪里无所谓，再接起来一张，比第一张小，放左边，
继续接，可能是中间数，就插在中间....依次

代码实现：

```
function insertSort(array) {

  let length = array.length;

  // 如果不是数组或者数组长度小于等于1，直接返回，不需要排序
  if (!Array.isArray(array) || length <= 1) return;

  // 循环从 1 开始，0 位置为默认的已排序的序列
  for (let i = 1; i < length; i++) {
    let temp = array[i]; // 保存当前需要排序的元素
    let j = i;

    // 在当前已排序序列中比较，如果比需要排序的元素大，就依次往后移动位置
    while (j -1 >= 0 && array[j - 1] > temp) {
      array[j] = array[j - 1];
      j--;
    }

    // 将找到的位置插入元素
    array[j] = temp;
```

```
  }

  return array;
}
```

当排序序列为已排序序列时，为最好的时间复杂度 O(n)。

插入排序的平均时间复杂度为 O(n²)，最坏时间复杂度为 O(n²)，空间复杂度为 O(1)，是稳定排序。

详细资料可以参考：
《图解排序算法(一)》

希尔排序

希尔排序的基本思想是把数组按下标的一定增量分组，对每组使用直接插入排序算法排序；随着增量逐渐减少，每组包含的元
素越来越多，当增量减至1时，整个数组恰被分成一组，算法便终止。

代码实现：

```
function hillSort(array) {

  let length = array.length;

  // 如果不是数组或者数组长度小于等于1，直接返回，不需要排序
  if (!Array.isArray(array) || length <= 1) return;


  // 第一层确定增量的大小，每次增量的大小减半
  for (let gap = parseInt(length >> 1); gap >= 1; gap = parseInt(gap >> 1)) {

    // 对每个分组使用插入排序，相当于将插入排序的1换成了 n
    for (let i = gap; i < length; i++) {
      let temp = array[i];
      let j = i;

      while (j - gap >= 0 && array[j - gap] > temp) {
        array[j] = array[j - gap];
        j -= gap;
      }
      array[j] = temp;
    }
  }

  return array;
}
```

希尔排序是利用了插入排序对于已排序序列排序效果最好的特点，在一开始序列为无序序列时，将序列分为多个小的分组进行
基数排序，由于排序基数小，每次基数排序的效果较好，然后在逐步增大增量，将分组的大小增大，由于每一次都是基于上一
次排序后的结果，所以每一次都可以看做是一个基本排序的序列，所以能够最大化插入排序的优点。

简单来说就是，由于开始时每组只有很少整数，所以排序很快。之后每组含有的整数越来越多，但是由于这些数也越来越有序，
所以排序速度也很快。

希尔排序的时间复杂度根据选择的增量序列不同而不同，但总的来说时间复杂度是小于 O(n^2) 的。

插入排序是一个稳定排序，但是在希尔排序中，由于相同的元素可能在不同的分组中，所以可能会造成相同元素位置的变化，
所以希尔排序是一个不稳定的排序。

希尔排序的平均时间复杂度为 O(nlogn) ，最坏时间复杂度为 O(n^s) ，空间复杂度为 O(1) ，不是稳定排序。

详细资料可以参考：
《图解排序算法(二)之希尔排序》
《数据结构基础 希尔排序 之 算法复杂度浅析》

归并排序

归并排序是利用归并的思想实现的排序方法，该算法采用经典的分治策略。递归的将数组两两分开直到只包含一个元素，然后
将数组排序合并，最终合并为排序好的数组。

代码实现：

```
function mergeSort(array) {

  let length = array.length;

  // 如果不是数组或者数组长度小于等于0，直接返回，不需要排序
  if (!Array.isArray(array) || length === 0) return;

  if (length === 1) {
    return array;
  }

  let mid = parseInt(length >> 1), // 找到中间索引值
    left = array.slice(0, mid), // 截取左半部分
    right = array.slice(mid, length); // 截取右半部分

  return merge(mergeSort(left), mergeSort(right)); // 递归分解后，进行排序合并
}


```

```
function merge(leftArray, rightArray) {

  let result = [],
    leftLength = leftArray.length,
    rightLength = rightArray.length,
    il = 0,
    ir = 0;

  // 左右两个数组的元素依次比较，将较小的元素加入结果数组中，直到其中一个数组的元素全部加入完则停止
  while (il < leftLength && ir < rightLength) {
    if (leftArray[il] < rightArray[ir]) {
      result.push(leftArray[il++]);
    } else {
      result.push(rightArray[ir++]);
    }
  }

  // 如果是左边数组还有剩余，则把剩余的元素全部加入到结果数组中。
  while (il < leftLength) {
    result.push(leftArray[il++]);
  }

  // 如果是右边数组还有剩余，则把剩余的元素全部加入到结果数组中。
  while (ir < rightLength) {
    result.push(rightArray[ir++]);
  }

  return result;
}
```

归并排序将整个排序序列看成一个二叉树进行分解，首先将树分解到每一个子节点，树的每一层都是一个归并排序的过程，每
一层归并的时间复杂度为 O(n)，因为整个树的高度为 lgn，所以归并排序的时间复杂度不管在什么情况下都为O(nlogn)。

归并排序的空间复杂度取决于递归的深度和用于归并时的临时数组，所以递归的深度为 logn，临时数组的大小为 n，所以归
并排序的空间复杂度为 O(n)。

归并排序的平均时间复杂度为 O(nlogn)，最坏时间复杂度为 O(nlogn)，空间复杂度为 O(n)，是稳定排序。

详细资料可以参考：
《图解排序算法(四)之归并排序》
《归并排序的空间复杂度？》

快速排序

快速排序的基本思想是通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据都比另外一部分的所有数据
都要小，然后再按此方法对这两部分数据分别进行快速排序，整个排序过程可以递归进行，以此达到整个数据变成有序序列。

代码实现：

```
function quickSort(array, start, end) {

  let length = array.length;

  // 如果不是数组或者数组长度小于等于1，直接返回，不需要排序
  if (!Array.isArray(array) || length <= 1 || start >= end) return;

  let index = partition(array, start, end); // 将数组划分为两部分，并返回右部分的第一个元素的索引值

  quickSort(array, start, index - 1); // 递归排序左半部分
  quickSort(array, index + 1, end); // 递归排序右半部分
}


function partition(array, start, end) {

  let pivot = array[start]; // 取第一个值为枢纽值，获取枢纽值的大小


  // 当 start 等于 end 指针时结束循环
  while (start < end) {

    // 当 end 指针指向的值大等于枢纽值时，end 指针向前移动
    while (array[end] >= pivot && start < end) {
      end--;
    }

    // 将比枢纽值小的值交换到 start 位置
    array[start] = array[end];

    // 移动 start 值，当 start 指针指向的值小于枢纽值时，start 指针向后移动
    while (array[start] < pivot && start < end) {
      start++;
    }

    // 将比枢纽值大的值交换到 end 位置，进入下一次循环
    array[end] = array[start];
  }

```

```
  // 将枢纽值交换到中间点
  array[start] = pivot;

  // 返回中间索引值
  return start;
}
```

这一种方法是填空法，首先将第一个位置的数作为枢纽值，然后 end 指针向前移动，当遇到比枢纽值小的值或者 end 值
等于 start 值的时候停止，然后将这个值填入 start 的位置，然后 start 指针向后移动，当遇到比枢纽值大的值或者
start 值等于 end 值的时候停止，然后将这个值填入 end 的位置。反复循环这个过程，直到 start 的值等于 end 的
值为止。将一开始保留的枢纽值填入这个位置，此时枢纽值左边的值都比枢纽值小，枢纽值右边的值都比枢纽值大。然后在递
归左右两边的的序列。

当每次换分的结果为含 ⌊n/2⌋和 ⌈n/2⌉−1 个元素时，最好情况发生，此时递归的次数为 logn，然后每次划分的时间复杂
度为 O(n)，所以最优的时间复杂度为 O(nlogn)。一般来说只要每次换分都是常比例的划分，时间复杂度都为 O(nlogn)。

当每次换分的结果为 n-1 和 0 个元素时，最坏情况发生。划分操作的时间复杂度为 O(n)，递归的次数为 n-1，所以最坏
的时间复杂度为 $O(n^2)$。所以当排序序列有序的时候，快速排序有可能被转换为冒泡排序。

快速排序的空间复杂度取决于递归的深度，所以最好的时候为 O(logn)，最坏的时候为 O(n)。

快速排序的平均时间复杂度为 O(nlogn) ，最坏时间复杂度为 $O(n^2)$ ，空间复杂度为 O(logn) ，不是稳定排序。

详细资料可以参考：
《图解排序算法(五)之快速排序——三数取中法》
《关于快速排序的四种写法》
《快速排序的时间和空间复杂度》
《快速排序最好，最坏，平均复杂度分析》
《快速排序算法的递归深度》

堆排序

堆排序的基本思想是：将待排序序列构造成一个大顶堆，此时，整个序列的最大值就是堆顶的根节点。将其与末尾元素进行
交换，此时末尾就为最大值。然后将剩余 n-1 个元素重新构造成一个堆，这样会得到 n 个元素的次小值。如此反复执行，
便能得到一个有序序列了。

```
function heapSort(array) {

  let length = array.length;
```

```
4
5   // 如果不是数组或者数组长度小于等于1，直接返回，不需要排序
6   if (!Array.isArray(array) || length <= 1) return;
7
8   buildMaxHeap(array); // 将传入的数组建立为大顶堆
9
10  // 每次循环，将最大的元素与末尾元素交换，然后剩下的元素重新构建为大顶堆
11  for (let i = length - 1; i > 0; i--) {
12    swap(array, 0, i);
13    adjustMaxHeap(array, 0, i); // 将剩下的元素重新构建为大顶堆
14  }
15
16  return array;
17 }
18
19
20 function adjustMaxHeap(array, index, heapSize) {
21   let iMax,
22     iLeft,
23     iRight;
24
25   while (true) {
26     iMax = index; // 保存最大值的索引
27     iLeft = 2 * index + 1; // 获取左子元素的索引
28     iRight = 2 * index + 2; // 获取右子元素的索引
29
30     // 如果左子元素存在，且左子元素大于最大值，则更新最大值索引
31     if (iLeft < heapSize && array[iMax] < array[iLeft]) {
32       iMax = iLeft;
33     }
34
35     // 如果右子元素存在，且右子元素大于最大值，则更新最大值索引
36     if (iRight < heapSize && array[iMax] < array[iRight]) {
37       iMax = iRight;
38     }
39
40     // 如果最大元素被更新了，则交换位置，使父节点大于它的子节点，同时将索引值跟新为被替
   换的值，继续检查它的子树
41     if (iMax !== index) {
42       swap(array, index, iMax);
43       index = iMax;
44     } else {
45
46       // 如果未被更新，说明该子树满足大顶堆的要求，退出循环
47       break;
48     }
49   }
50 }
51
```

```
// 构建大顶堆
function buildMaxHeap(array) {
  let length = array.length,
    iParent = parseInt(length >> 1) - 1; // 获取最后一个非叶子点的元素

  for (let i = iParent; i >= 0; i--) {
    adjustMaxHeap(array, i, length); // 循环调整每一个子树，使其满足大顶堆的要求
  }
}

// 交换数组中两个元素的位置
function swap(array, i, j) {
  let temp = array[i];
  array[i] = array[j];
  array[j] = temp;
}
```

建立堆的时间复杂度为 O(n)，排序循环的次数为 n-1，每次调整堆的时间复杂度为 O(logn)，因此堆排序的时间复杂度在
不管什么情况下都是 O(nlogn)。

堆排序的平均时间复杂度为 O(nlogn)，最坏时间复杂度为 O(nlogn)，空间复杂度为 O(1)，不是稳定排序。

详细资料可以参考：
《图解排序算法(三)之堆排序》
《常见排序算法 - 堆排序 (Heap Sort)》
《堆排序中建堆过程时间复杂度O(n)怎么来的？》
《排序算法之 堆排序 及其时间复杂度和空间复杂度》
《最小堆 构建、插入、删除的过程图解》

基数排序

基数排序是一种非比较型整数排序算法，其原理是将整数按位数切割成不同的数字，然后按每个位数分别比较。排序过程：将
所有待比较数值（正整数）统一为同样的数位长度，数位较短的数前面补零。然后，从最低位开始，依次进行一次排序。这样
从最低位排序一直到最高位排序完成以后，数列就变成一个有序序列。

代码实现：

```
function radixSort(array) {

  let length = array.length;

  // 如果不是数组或者数组长度小于等于1，直接返回，不需要排序
  if (!Array.isArray(array) || length <= 1) return;

  let bucket = [],
```

```
    max = array[0],
    loop;

  // 确定排序数组中的最大值
  for (let i = 1; i < length; i++) {
    if (array[i] > max) {
      max = array[i];
    }
  }

  // 确定最大值的位数
  loop = (max + '').length;


  // 初始化桶
  for (let i = 0; i < 10; i++) {
    bucket[i] = [];
  }

  for (let i = 0; i < loop; i++) {
    for (let j = 0; j < length; j++) {
      let str = array[j] + '';

      if (str.length >= i + 1) {
        let k = parseInt(str[str.length - 1 - i]); // 获取当前位的值，作为插入的索引
        bucket[k].push(array[j]);
      } else {
        // 处理位数不够的情况，高位默认为 0
        bucket[0].push(array[j]);
      }
    }

    array.splice(0, length); // 清空旧的数组

    // 使用桶重新初始化数组
    for (let i = 0; i < 10; i++) {
      let t = bucket[i].length;

      for (let j = 0; j < t; j++) {
        array.push(bucket[i][j]);
      }

      bucket[i] = [];
    }
  }

  return array;

```

```
57  }
```

基数排序的平均时间复杂度为 O(nk)，k 为最大元素的长度，最坏时间复杂度为 O(nk)，空间复杂度为 O(n) ，是稳定
排序。

详细资料可以参考：
《常见排序算法 - 基数排序》
《排序算法之 基数排序 及其时间复杂度和空间复杂度》

算法总结可以参考：
《算法的时间复杂度和空间复杂度-总结》
《十大经典排序算法（动图演示）》
《各类排序算法的对比及实现》

快速排序相对于其他排序效率更高的原因

上面一共提到了8种排序的方法，在实际使用中，应用最广泛的是快速排序。快速排序相对于其他排序算法的优势在于在相同
数据量的情况下，它的运算效率最高，并且它额外所需空间最小。

我们首先从时间复杂度来判断，由于前面几种方法的时间复杂度平均情况下基本趋向于 $O(n^2)$，因此只从时间复杂度上来看
的话，显然归并排序、堆排序和快速排序的时间复杂度最小。但是既然这几种方法的时间复杂度基本一致，并且快速排序在最
坏情况下时间的复杂度还会变为 $O(n^2)$，那么为什么它的效率反而更高呢?

首先在对大数据量排序的时候，由于归并排序的空间复杂度为 O(n)，因此归并排序在这种情况下会需要过多的额外内存，因
此归并排序首先就被排除掉了。

接下来就剩下了堆排序和快速排序的比较。我认为堆排序相对于快速排序的效率不高的原因有两个方面。

第一个方面是对于比较操作的有效性来说。对于快速排序来说，每一次元素的比较都会确定该元素在数组中的位置，也就是在
枢纽值的左边或者右边，快速排序的每一次比较操作都是有意义的结果。而对于堆排序来说，在每一次重新调整堆的时候，我
们在迭代时，已经知道上层的节点值一定比下层的节点值大，因此当我们每次为了打乱堆结构而将最后一个元素与堆顶元素互
换时，互换后的元素一定是比下层元素小的，因此我们知道比较结果却还要在堆结构调整时去进行再一次的比较，这样的比较
是没有意义的，以此在堆排序中会产生大量的没有意义的比较操作。

第二个方面是对于缓存局部性原理的利用上来考虑的，我认为这应该是造成堆排序的效率不如快速排序的主要原因。在计算机
中利用了多级缓存的机制，来解决 cpu 计算速度与存储器数据读取速度间差距过大的问题。缓存的原理主要是基于局部性原
理，局部性原理简单来说就是，当前被访问过的数据，很有可能在一段时间内被再次访问，这被称为时间局部性。还有就是当

前访问的数据，那么它相邻的数据，也有可能在一段时间内被访问到，这被称为空间局部性。计算机缓存利用了局部性的原理
来对数据进行缓存，来尽可能少的减少磁盘的 I/O 次数，以此来提高执行效率。对于堆排序来说，它最大的问题就是它对于
空间局部性的违背，它在进行比较时，比较的并不是相邻的元素，而是与自己相隔很远的元素，这对于利用空间局部性来进行
数据缓存的计算机来说，它的很多缓存都是无效的。并且对于大数据量的排序来说，缓存的命中率就会变得很低，因此会明显
提高磁盘的 I/O 次数，并且由于堆排序的大量的无效比较，因此这样就造成了堆排序执行效率的低下。而相对来快速排序来
说，它的排序每一次都是在相邻范围内的比较，并且比较的范围越来越小，它很好的利用了局部性原理，因此它的执行效率更
高。简单来说就是在堆排序中获取一个元素的值所花费的时间比在快速排序获取一个元素的值所花费的时间要大。因此我们可
以看出，时间复杂度类似的算法，在计算机中实际执行可能会有很大的差别，因为决定算法执行效率的还有内存读取这样的其
他的因素。

相关资料可以参考：
《为什么在平均情况下快速排序比堆排序要优秀？》
《为什么说快速排序是性能最好的排序算法？》

系统自带排序实现

每个语言的排序内部实现都是不同的。

对于 JS 来说，数组长度大于 10 会采用快排，否则使用插入排序。选择插入排序是因为虽然时间复杂度很差，但是在数据
量很小的情况下和 O(N * logN) 相差无几，然而插入排序需要的常数时间很小，所以相对别的排序来说更快。

稳定性

稳定性的意思就是对于相同值来说，相对顺序不能改变。通俗的讲有两个相同的数 A 和 B，在排序之前 A 在 B 的前面，
而经过排序之后，B 跑到了 A 的前面，对于这种情况的发生，我们管他叫做排序的不稳定性。

稳定性有什么意义？个人理解对于前端来说，比如我们熟知框架中的虚拟 DOM 的比较，我们对一个 `<ul>` 列表进行渲染，
当数据改变后需要比较变化时，不稳定排序或操作将会使本身不需要变化的东西变化，导致重新渲染，带来性能的损耗。

排序面试题目总结

1. 快速排序在完全无序的情况下效果最好，时间复杂度为O(nlogn)，在有序情况下效果最差，时间复杂度为O(n^2)。
2. 初始数据集的排列顺序对算法的性能无影响的有堆排序，直接选择排序，归并排序，基数排序。
3. 合并 m 个长度为 n 的已排序数组的时间复杂度为 O(nmlogm)。
4. 外部排序常用的算法是归并排序。

5. 数组元素基本有序的情况下，插入排序效果最好，因为这样只需要比较大小，不需要移动，时间复杂度趋近于O(n)。
6. 如果只想得到1000个元素组成的序列中第5个最小元素之前的部分排序的序列，用堆排序方法最快。
7. 插入排序和优化后的冒泡在最优情况（有序）都只用比较 n-1 次。
8. 对长度为 n 的线性表作快速排序，在最坏情况下，比较次数为 n(n-1)/2。
9. 下标从1开始，在含有 n 个关键字的小根堆（堆顶元素最小）中，关键字最大的记录有可能存储在 [n/2]+2 位置上。
   因为小根堆中最大的数一定是放在叶子节点上，堆本身是个完全二叉树，完全二叉树的叶子节点的位置大于 [n/2]。
10. 拓扑排序的算法，每次都选择入度为0的结点从图中删去，并从图中删除该顶点和所有以它为起点的有向边。
11. 任何一个基于"比较"的内部排序的算法，若对 n 个元素进行排序，则在最坏情况下所需的比较次数 k 满足 2^k > n!，
    时间下界为 O(nlogn)
12. m 个元素 k 路归并的归并趟数 s=logk(m)，代入数据：logk(100)≦3
13. 对 n 个记录的线性表进行快速排序为减少算法的递归深度，每次分区后，先处理较短的部分。
14. 在用邻接表表示图时，拓扑排序算法时间复杂度为 O(n+e)

树

二叉树相关性质

1. 节点的度：一个节点含有的子树的个数称为该节点的度；
2. 叶节点或终端节点：度为零的节点；
3. 节点的层次：从根开始定义起，根为第1层，根的子节点为第2层，以此类推。
4. 树的高度或深度：树中节点的最大层次。
5. 在非空二叉树中，第 i 层的结点总数不超过 2^(i-1)，i>=1。
6. 深度为 h 的二叉树最多有 2^h-1个结点(h>=1)，最少有 h 个结点。
7. 对于任意一棵二叉树，如果其叶结点数为 N0，而度数为2的结点总数为 N2，则 N0 = N2+1；
8. 给定 N 个节点，能构成 h(N) 种不同的二叉树。h(N)为卡特兰数的第 N 项。(2n)!/(n!(n+1)!)。
9. 二叉树的前序遍历，首先访问根结点，然后遍历左子树，最后遍历右子树。简记根-左-右。
10. 二叉树的中序遍历，首先遍历左子树，然后访问根结点，最后遍历右子树。简记左-根-右。
11. 二叉树的后序遍历，首先遍历左子树，然后遍历右子树，最后访问根结点。简记左-右-根。
12. 二叉树是非线性数据结构，但是顺序存储结构和链式存储结构都能存储。
13. 一个带权的无向连通图的最小生成树的权值之和是唯一的。
14. 只有一个结点的二叉树的度为 0 。
15. 二叉树的度是以节点的最大的度数定义的。
16. 树的后序遍历序列等同于该树对应的二叉树的中序序列。
17. 树的先序遍历序列等同于该树对应的二叉树的先序序列。
18. 线索二叉树的线索实际上指向的是相应遍历序列特定结点的前驱结点和后继结点，所以先写出二叉树的中序遍历序列：
    debxac，中序遍历中在x左边和右边的字符，就是它在中序线索化的左、右线索，即 b、a 。
19. 递归式的先序遍历一个 n 节点，深度为 d 的二叉树，需要栈空间的大小为 O（d），因为二叉树并不一定是平衡的，
    也就是深度d！=logn，有可能d>>logn。所以栈大小应该是O（d）
20. 一棵具有 N 个结点的二叉树的前序序列和后序序列正好相反 ，则该二叉树一定满足该二叉树只有

左子树或只有右子树，
即该二叉树一定是一条链（二叉树的高度为N，高度等于结点数）。

21. 引入二叉线索树的目的是加快查找结点的前驱或后继的速度。
22. 二叉树线索化后，先序线索化与后序线索化最多有1个空指针域，而中序线索化最多有2个空指针域。
23. 不管是几叉树，节点数等于=分叉数+1
24. 任何一棵二叉树的叶子结点在先序、中序和后序遍历中的相对次序不发生改变。

详细资料可以参考：
《n 个节点的二叉树有多少种形态》
《数据结构二叉树知识点总结》
《还原二叉树--已知先序中序或者后序中序》
《树、森林与二叉树的转换》

满二叉树

对于一棵二叉树，如果每一个非叶子节点都存在左右子树，并且二叉树中所有的叶子节点都在同一层中，这样的二叉树称为满
二叉树。

完全二叉树

对于一棵具有 n 个节点的二叉树按照层次编号，同时，左右子树按照先左后右编号，如果编号为 i 的节点与同样深度的满
二叉树中编号为i的节点在满二叉树中的位置完全相同，则这棵二叉树称为完全二叉树。

性质：

1. 具有 n 个结点的完全二叉树的深度为 K =[log2n」+1(取下整数)
2. 有 N 个结点的完全二叉树各结点如果用顺序方式存储，则结点之间有如下关系： 若 I 为结点编号（从1开始编号）则
   如果 I>1，则其父结点的编号为 I/2；
3. 完全二叉树，如果 2 * I <= N，则其左儿子（即左子树的根结点）的编号为2 * I；若2 * I > N，则无左儿子；如
   果 2 * I + 1 <= N，则其右儿子的结点编号为 2 * I + 1；若 2 * I + 1 > N，则无右儿子。

平衡二叉查找树（AVL）

平衡二叉查找树具有如下几个性质：

1. 可以是空树。
2. 假如不是空树，任何一个结点的左子树与右子树都是平衡二叉树，并且高度之差的绝对值不超过1。

平衡二叉树是为了解决二叉查找树中出现链式结构（只有左子树或只有右子树）的情况，这样的情况出现后对我们的查找没有
一点帮帮助，反而增加了维护的成本。

平衡因子使用两个字母来表示。第一个字母表示最小不平衡子树根结点的平衡因子，第二个字母表示最小不平衡子树较高子树
的根结点的平衡因子。根据不同的情况使用不同的方法来调整失衡的子树。

详细资料可以参考：
《平衡二叉树，AVL树之图解篇》

B-树

B-树主要用于文件系统以及部分数据库索引，如 MongoDB。使用 B-树来作为数据库的索引主要是为了减少查找是磁盘的 I/O
次数。试想，如果我们使用二叉查找树来作为索引，那么查找次数的最坏情况等于二叉查找树的高度，由于索引存储在磁盘中，
我们每次都只能加载对应索引的磁盘页进入内存中比较，那么磁盘的 I/O 次数就等于索引树的高度。所以采用一种办法来减少
索引树的高度是提高索引效率的关键。

B-树是一种多路平衡查找树，它的每一个节点最多包含 K 个子节点，K 被称为 B-树的阶，K 的大小取决于磁盘页的大小。每
个节点中的元素从小到大排列，节点当中 k-1 个元素正好是 k 个孩子包含的元素的值域分划。简单来说就是以前一个磁盘页
只存储一个索引的值，但 B-树中一个磁盘页中存储了多个索引的值，因此在相同的比较范围内，B-树相对于一般的二叉查找树
的高度更小。其实它的主要目的就是每次尽可能多的将索引值加载入内存中进行比较，以此来减少磁盘的 I/O 次数，其实就查
找次数而言，和二叉查找树比较差不了多少，只是说这个比较过程是在内存中完成的，速度更快而已。

详细资料可以参考：
《漫画：什么是 B- 树？》

B+树

B+ 树相对于 B-树有着更好的查找性能，根据 B-树我们可以知道，要想加快索引速度的方法就是尽量减少磁盘 I/O 的次数。
B+ 树相对于 B-的主要变化是，每个中间节点中不再包含卫星数据，只有叶子节点包含卫星数据，每个父节点都出现在子节点
中，叶子节点依次相连，形成一个顺序链表。中间节点不包含卫星数据，只用来作为索引使用，这意味着每一个磁盘页中能够
包含更多的索引值。因此 B+ 树的高度相对于 B-来说更低，所以磁盘的 I/O 次数更少。由于叶子节点依次相连，并且包含
了父节点，所以可以通过叶子节点来找到对应的值。同时 B+ 树所有查询都要查找到叶子节点，查询性能比 B-树稳定。

详细资料可以参考：
《漫画：什么是 B+ 树？》

数据库索引

数据库以 B 树或者 B+ 树格式来储存的数据的，一张表是根据主键来构建的树的结构。因此如果想查找其他字段，就需要建
立索引，我对于索引的理解是它就是以某个字段为关键字的 B 树文件，通过这个 B 树文件就能够提高数据查找的效率。但是
由于我们需要维护的是平衡树的结构，因此对于数据的写入、修改、删除就会变慢，因为这有可能会涉及到树的平衡调整。

相关资料可以参考：
《深入浅出数据库索引原理》
《数据库的最简单实现》

红黑树

红黑树是一种自平衡的二叉查找树，它主要是为了解决不平衡的二叉查找树的查找效率不高的缺点。红黑树保证了从根到叶子
节点的最长路径不会超过最短路径的两倍。

红黑树的有具体的规则：

1.节点是红色或黑色。

2.根节点是黑色。

3.每个叶子节点都是黑色的空节点（NIL节点）。

4 每个红色节点的两个子节点都是黑色。(从每个叶子到根的所有路径上不能有两个连续的红色节点)

5.从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点。

当红黑树发生删除和插入导致红黑树不满足这些规则时，需要通过处理，使其重新满足这些规则。

详细资料可以参考：
《漫画：什么是红黑树？》
《漫画算法等精选文章目录》

Huffman 树

给定 n 权值作为 n 个叶子节点，构造一棵二叉树，若这棵二叉树的带权路径长度达到最小，则称这样的二叉树为最优二叉
树，也称为 Huffman 树。

利用 Huffman 树对每一个字符编码，该编码又称为 Huffman 编码，Huffman 编码是一种前缀编码，即一个字符的编码
不是另一个字符编码的前缀。

性质：

1. 对应一组权重构造出来的 Huffman 树一般不是唯一的
2. Huffman 树具有最小的带权路径长度
3. Huffman 树中没有度为1的结点
4. 哈夫曼树是带权路径长度最短的树，路径上权值较大的结点离根较近

5. Huffman 树的带权路径长度 WPL 等于各叶子结点的带权路径长度之和

详细资料可以参考：

《数据结构和算法—— Huffman 树和 Huffman 编码》
《详细图解哈夫曼 Huffman 编码树》

二叉查找树

二叉查找树是一种特殊的二叉树，相对较小的值保存在左节点中，较大的值保存在右节点中，这一特性使得查找的效率很高，
对于数值型和非数值型数据，比如字母和字符串，都是如此。

实现树节点类：

```
// 节点类，树的节点
class Node {
  constructor(value) {
    this.value = value;
    this.left = null;
    this.right = null;
  }

  show() {
    console.log(this.value);
  }
}
```

实现二叉查找树类：

```
class BinarySearchTree {

  constructor() {
    this.root = null
  }

}
```

实现树的节点插入方法

节点插入的基本思想是将插入节点和当前节点做比较，如果比当前节点值小，并且没有左子树，那么将节点作为左叶子节点，
否则继续和左子树进行比较。如果比当前节点值大，并且没有右子树，则将节点作为右叶子节点，否则继续和右子树进行比较。
循环这个过程直到找到合适的插入位置。

```

  insert(value) {

```

```
    let newNode = new Node(value);

    // 判断根节点是否为空，如果不为空则递归插入到树中
    if (this.root === null) {
      this.root = newNode;
    } else {
      this.insertNode(this.root, newNode);
    }
  }

  insertNode(node, newNode) {

    // 将插入节点的值与当前节点的进行比较，如果比当前节点小，则递归判断左子树，如果比当前节点大，则递归判断右子树。
    if (newNode.value < node.value) {
      if (node.left === null) {
        node.left = newNode;
      } else {
        this.insertNode(node.left, newNode);
      }
    } else {
      if (node.right === null) {
        node.right = newNode;
      } else {
        this.insertNode(node.right, newNode);
      }
    }

  }
```

通过递归实现树的先序、中序、后序遍历

```
 // 先序遍历通过递归实现
 // 先序遍历则先打印当前节点，再递归打印左子节点和右子节点。
  preOrderTraverse() {
    this.preOrderTraverseNode(this.root);
  }

  preOrderTraverseNode(node) {
    if (node !== null) {
      node.show();
      this.preOrderTraverseNode(node.left);
      this.preOrderTraverseNode(node.right);
    }
  }

  // 中序遍历通过递归实现
  // 中序遍历则先递归打印左子节点，再打印当前节点，最后再递归打印右子节点。
```

```
  inOrderTraverse() {
    this.inOrderTraverseNode(this.root);
  }

  inOrderTraverseNode(node) {
    if (node !== null) {
      this.inOrderTraverseNode(node.left);
      node.show();
      this.inOrderTraverseNode(node.right);
    }
  }

  // 后序遍历通过递归实现
  // 后序遍历则先递归打印左子节点和右子节点，最后再打印当前节点。
  postOrderTraverse() {
    this.postOrderTraverseNode(this.root);
  }

  postOrderTraverseNode(node) {
    if (node !== null) {
      this.postOrderTraverseNode(node.left);
      this.postOrderTraverseNode(node.right);
      node.show();
    }
  }
```

通过循环实现树的先序、中序、后序遍历

```
  // 先序遍历通过循环实现
  // 通过栈来实现循环先序遍历，首先将根节点入栈。然后进入循环，每次循环开始，当前节点出栈，打印当前节点，然后将
  // 右子节点入栈，再将左子节点入栈，然后进入下一循环，直到栈为空结束循环。
  preOrderTraverseByStack() {
    let stack = [];

    // 现将根节点入栈，开始遍历
    stack.push(this.root);

    while (stack.length > 0) {

      // 从栈中获取当前节点
      let node = stack.pop();

      // 执行节点操作
      node.show();

      // 判断节点是否还有左右子节点，如果存在则加入栈中，注意，由于中序遍历先序遍历是先访问根
```

```
      // 再访问左和右子节点，因此左右子节点的入栈顺序应该是反过来的
      if (node.right) {
        stack.push(node.right);
      }

      if (node.left) {
        stack.push(node.left);
      }
    }
  }

  // 中序遍历通过循环实现
  // 中序遍历先将所有的左子节点入栈，如果左子节点为 null 时，打印栈顶元素，然后判断该元素是否有右子树，如果有
  // 则将右子树作为起点重复上面的过程，一直循环直到栈为空并且节点为空时。
  inOrderTraverseByStack() {
    let stack = [],
      node = this.root;

    // 中序遍历是先左再根最后右
    // 所以首先应该先把最左边节点遍历到底依次 push 进栈
    // 当左边没有节点时，就打印栈顶元素，然后寻找右节点
    while (stack.length > 0 || node) {
      if (node) {
        stack.push(node);
        node = node.left;
      } else {
        node = stack.pop();
        node.show();
        node = node.right;
      }
    }
  }

  // 后序遍历通过循环来实现
  // 使用两个栈来是实现，先将根节点放入栈1中，然后进入循环，每次循环将栈顶元素加入栈2，再依次将左节点和右节点依次
  // 加入栈1中，然后进入下一次循环，直到栈1的长度为0。最后再循环打印栈2的值。
  postOrderTraverseByStack() {
    let stack1 = [],
      stack2 = [],
      node = null;

    // 后序遍历是先左再右最后根
    // 所以对于一个栈来说，应该先 push 根节点
    // 然后 push 右节点，最后 push 左节点

    stack1.push(this.root);

```

```
    while (stack1.length > 0) {
      node = stack1.pop();

      stack2.push(node);

      if (node.left) {
        stack1.push(node.left);
      }

      if (node.right) {
        stack1.push(node.right);
      }

    }

    while (stack2.length > 0) {
      node = stack2.pop();
      node.show();
    }
  }
```

实现寻找最大最小节点值

```
// 寻找最小值，在最左边的叶子节点上
 findMinNode(root) {
    let node = root;

    while (node && node.left) {
      node = node.left;
    }

    return node;
  }

  // 寻找最大值，在最右边的叶子节点上

  findMaxNode(root) {
    let node = root;

    while (node && node.right) {
      node = node.right;
    }

    return node;
  }
```

实现寻找特定大小节点值

```
// 寻找特定值
find(value) {
  return this.findNode(this.root, value);
}

findNode(node, value) {

  if (node === null) {
    return node;
  }
  if (value < node.value) {
    return this.findNode(node.left, value);
  } else if (value > node.value) {
    return this.findNode(node.right, value);
  } else {
    return node;
  }
}
```

实现移除节点值

移除节点的基本思想是，首先找到需要移除的节点的位置，然后判断该节点是否有叶节点。如果没有叶节点，则直接删除，如
果有一个叶子节点，则用这个叶子节点替换当前的位置。如果有两个叶子节点，则去右子树中找到最小的节点来替换当前节点。

```

// 移除指定值节点
remove(value) {
  this.removeNode(this.root, value);
}
removeNode(node, value) {

  if (node === null) {
    return node;
  }

  // 寻找指定节点
  if (value < node.value) {
    node.left = this.removeNode(node.left, value);
    return node;
  } else if (value > node.value) {
    node.right = this.removeNode(node.right, value);
    return node;
  } else { // 找到节点

    // 第一种情况—没有叶节点
    if (node.left === null && node.right === null) {
```

```
        node = null;
        return node;
      }

      // 第二种情况——一个只有一个子节点的节点，将节点替换为节点的子节点
      if (node.left === null) {
        node = node.right;
        return node;
      } else if (node.right === null) {
        node = node.left;
      }

      // 第三种情况——一个有两个子节点的节点，去右子树中找到最小的节点，用它的值来替换当前节点
      // 的值，保持树的特性，然后将替换的节点去掉
      let aux = this.findMinNode(node.right);
      node.value = aux.value;
      node.right = this.removeNode(node.right, aux);
      return node;
    }
  }
```

求解二叉树中两个节点的最近公共祖先节点

```
求解二叉树中的两个节点的最近公共祖先节点可以分为三种情况来考虑

（1）该二叉树为搜索二叉树

    解决办法，首先从根节点开始遍历。如果根节点的值比两个节点的值都大的情况下，则说明两个节点的共同祖先存在于
    根节点的左子树中，因此递归遍历左子树。反之，则遍历右子树。当当前节点的值比其中一个节点的值大，比其中一个
    节点的值小时，该节点则为两个节点的最近公共祖先节点。

（2）该二叉树为普通二叉树，但是每个节点含有指向父节点的指针。

    通过指向父节点的指针，我们可以通过节点得到它的所有父节点，该父节点列表可以看做是一个链表，因此求两个节点
    的最近公共祖先节点就可以看做是求两个链表的最近公共节点，以此来找到两个节点的最近公共祖先节点。

（3）该二叉树为普通二叉树，节点不含有指向父节点的指针。

    这种情况下，我们可以从根节点出发，分别得到根节点到两个节点的路径。然后遍历两条路径，直到遇到第一个不相同
    的节点为止，这个时候该节点前面的那个节点则为两个节点的最近公共祖先节点。
```

详细资料可以参考：
《二叉树中两个节点的最近公共祖先节点》

链表

反转单向链表

需要将一个单向链表反转。思路很简单，使用三个变量分别表示当前节点和当前节点的前后节点，虽然这题很简单，但是却是
一道面试常考题。

思路是从头节点往后遍历，先获取下一个节点，然后将当前节点的 next 设置为前一个节点，然后再继续循环。

```
var reverseList = function(head) {
    // 判断下变量边界问题
    if (!head || !head.next) return head;
    // 初始设置为空，因为第一个节点反转后就是尾部，尾部节点指向 null
    let pre = null;
    let current = head;
    let next;
    // 判断当前节点是否为空
    // 不为空就先获取当前节点的下一节点
    // 然后把当前节点的 next 设为上一个节点
    // 然后把 current 设为下一个节点，pre 设为当前节点
    while(current) {
        next = current.next;
        current.next = pre;
        pre = current;
        current = next;
    }
    return pre;
};
```

动态规划

爬楼梯问题

有一座高度是10级台阶的楼梯，从下往上走，每跨一步只能向上1级或者2级台阶。要求用程序来求出一共有多少种走法?

递归方法分析

由分析可知，假设我们只差最后一步就能走上第10级阶梯，这个时候一共有两种情况，因为每一步只允许走1级或2级阶梯，
因此分别为从8级阶梯和从9九级阶梯走上去的情况。因此从0到10级阶梯的走法数量就等于从0到9级阶梯的走法数量加上
从0到8级阶梯的走法数量。依次类推，我们可以得到一个递归关系，递归结束的标志为从0到1级阶梯的

走法数量和从0到
2级阶梯的走法数量。

代码实现

```
function getClimbingWays(n) {

  if (n < 1) {
    return 0;
  }

  if (n === 1) {
    return 1;
  }

  if (n === 2) {
    return 2;
  }

  return getClimbingWays(n - 1) + getClimbingWays(n - 2);
}
```

使用这种方法时整个的递归过程是一个二叉树的结构，因此该方法的时间复杂度可以近似的看为
O(2^n)，空间复杂度
为递归的深度 O(logn)。

备忘录方法

分析递归的方法我们可以发现，其实有很多的计算过程其实是重复的，因此我们可以使用一个数组，将已经计算出的值给
保存下来，每次计算时，先判断计算结果是否已经存在，如果已经存在就直接使用。

代码实现

```
let map = new Map();

function getClimbingWays(n) {

  if (n < 1) {
    return 0;
  }

  if (n === 1) {
    return 1;
  }

  if (n === 2) {
    return 2;
  }
```

```
16
17    if (map.has(n)) {
18      return map.get(n);
19    } else {
20      let value = getClimbingWays(n - 1) + getClimbingWays(n - 2);
21      map.set(n, value);
22      return value;
23    }
24  }
```

通过这种方式，我们将算法的时间复杂度降低为 O(n)，但是增加空间复杂度为 O(n)

迭代法

通过观察，我们可以发现每一个值其实都等于它的前面两个值的和，因此我们可以使用自底向上的方式来实现。

代码实现

```
1   function getClimbingWays(n) {
2
3     if (n < 1) {
4       return 0;
5     }
6
7     if (n === 1) {
8       return 1;
9     }
10
11    if (n === 2) {
12      return 2;
13    }
14
15    let a = 1,
16      b = 2,
17      temp = 0;
18
19    for (let i = 3; i <= n; i++) {
20      temp = a + b;
21      a = b;
22      b = temp;
23    }
24
25    return temp;
26  }
```

通过这种方式我们可以将算法的时间复杂度降低为 O(n)，并且将算法的空间复杂度降低为 O(1)。

详细资料可以参考：
《漫画：什么是动态规划？（整合版）》

经典笔试题

1. js 实现一个函数，完成超过范围的两个大整数相加功能

```
主要思路是通过将数字转换为字符串，然后每个字符串在按位相加。

function bigNumberAdd(number1, number2) {

  let result = "", // 保存最后结果
    carry = false; // 保留进位结果

  // 将字符串转换为数组
  number1 = number1.split("");
  number2 = number2.split("");

  // 当数组的长度都变为0，并且最终不再进位时，结束循环
  while (number1.length || number2.length || carry) {

    // 每次将最后的数字进行相加，使用~~的好处是，即使返回值为 undefined 也能转换为0
    carry += ~~number1.pop() + ~~number2.pop();

    // 取加法结果的个位加入最终结果
    result = carry % 10 + result;

    // 判断是否需要进位，true 和 false 的值在加法中会被转换为 1 和 0
    carry = carry > 9;
  }

  // 返回最终结果
  return result;
}
```

详细资料可以参考：
《JavaScript实现超范围的数相加》
《js 实现大整数加法》

2. js 如何实现数组扁平化?

```
// 这一种方法通过递归来实现，当元素为数组时递归调用，兼容性好
function flattenArray(array) {

  if (!Array.isArray(array)) return;

  let result = [];
```

```
    result = array.reduce(function (pre, item) {
      // 判断元素是否为数组，如果为数组则递归调用，如果不是则加入结果数组中
      return pre.concat(Array.isArray(item) ? flattenArray(item) : item);
    }, []);

    return result;
  }

  // 这一种方法是利用了 toString 方法，它的一个缺点是改变了元素的类型，只适合于数组中元素都是整数的情况
  function flattenArray(array) {
    return array.toString().split(",").map(function (item) {
      return +item;
    })
  }
```

详细资料可以参考：
《JavaScript专题之数组扁平化》

3. js 如何实现数组去重?

```
  function unique(array) {
    if (!Array.isArray(array) || array.length <= 1) return;

    var result = [];

    array.forEach(function (item) {
      if (result.indexOf(item) === -1) {
        result.push(item);
      }
    })

    return result;
  }


  function unique(array) {
    if (!Array.isArray(array) || array.length <= 1) return;

    return [...new Set(array)];
  }
```

详细资料可以参考：
《JavaScript专题之数组去重》

4. 如何求数组的最大值和最小值?

```
    var arr = [6, 4, 1, 8, 2, 11, 23];
    console.log(Math.max.apply(null, arr))
```

详细资料可以参考：
《JavaScript专题之如何求数组的最大值和最小值》

5. 如何求两个数的最大公约数?

```
    基本思想是采用辗转相除的方法，用大的数去除以小的那个数，然后再用小的数去除以的得到的余数，一直这样递归下去，
    直到余数为0时，最后的被除数就是两个数的最大公约数。

    function getMaxCommonDivisor(a, b) {
      if (b === 0) return a;

      return getMaxCommonDivisor(b, a % b);
    }
```

6. 如何求两个数的最小公倍数?

```
    基本思想是采用将两个数相乘，然后除以它们的最大公约数

    function getMinCommonMultiple(a, b){
      return a * b / getMaxCommonDivisor(a, b);
    }
```

详细资料可以参考：
《百度 web 前端面试题之求两个数的最大公约数和最小公倍数》

7. 实现 IndexOf 方法?

```
   function indexFun(array, val) {
     if (!Array.isArray(array)) return;

     let length = array.length;

     for (let i = 0; i < length; i++) {
       if (array[i] === val) {
         return i;
       }
     }

     return -1;
   }
```

8. 判断一个字符串是否为回文字符串?

```
   function isPalindrome(str) {
     let reg = /[\W_]/g, // 匹配所有非单词的字符以及下划线
       newStr = str.replace(reg, "").toLowerCase(), // 替换为空字符并将大写字母转换为小写
       reverseStr = newStr.split("").reverse().join(""); // 将字符串反转

     return reverseStr === newStr;
   }
```

9. 实现一个累加函数的功能比如 sum(1,2,3)(2).valueOf()

```
   function sum(...args) {

   let result = 0;

   result = args.reduce(function (pre, item) {
     return pre + item;
   }, 0);

   let add = function (...args) {

     result = args.reduce(function (pre, item) {
       return pre + item;
     }, result);

     return add;
   };

   add.valueOf = function () {
     console.log(result);
   }

   return add;
   }
```

10. 使用 reduce 方法实现 forEach、map、filter

```

   // forEach
   function forEachUseReduce(array, handler) {
     array.reduce(function (pre, item, index) {
       handler(item, index);
     });
   }

```

```
// map
function mapUseReduce(array, handler) {
  let result = [];

  array.reduce(function (pre, item, index) {
    let mapItem = handler(item, index);
    result.push(mapItem);
  });

  return result;
}

// filter
function filterUseReduce(array, handler) {
  let result = [];

  array.reduce(function (pre, item, index) {
    if (handler(item, index)) {
      result.push(item);
    }
  });

  return result;
}
```

11. 设计一个简单的任务队列，要求分别在 1,3,4 秒后打印出 "1", "2", "3"

```
class Queue {

  constructor() {
    this.queue = [];
    this.time = 0;
  }

  addTask(task, t) {
    this.time += t;
    this.queue.push([task, this.time]);
    return this;
  }

  start() {
    this.queue.forEach(item => {
      setTimeout(() => {
        item[0]();
      }, item[1]);
    })
  }
}
```

12. 如何查找一篇英文文章中出现频率最高的单词?

```
function findMostWord(article) {

// 合法性判断
if (!article) return;

// 参数处理
article = article.trim().toLowerCase();

let wordList = article.match(/[a-z]+/g),
 visited = [],
 maxNum = 0,
 maxWord = "";

article = " " + wordList.join("  ") + " ";

// 遍历判断单词出现次数
wordList.forEach(function (item) {
 if (visited.indexOf(item) < 0) {
   let word = new RegExp(" " + item + " ", "g"),
     num = article.match(word).length;

   if (num > maxNum) {
     maxNum = num;
     maxWord = item;
   }
 }
});

return maxWord + "  " + maxNum;

}
```

常见面试智力题总结

1. 时针与分针夹角度数问题?

分析:

```
    当时间为 m 点 n 分时，其时针与分针夹角的度数为多少?

    我们可以这样考虑，分针每走一格为 6 度，分针每走一格对应的时针会走 0.5 度。

    时针每走一格为 30 度。

    因此，时针走过的度数为 m * 30 + n * 0.5，分针走过的度数为 n * 6。

    因此时针与分针的夹角度数为 |m * 30 + n * 0.5 - n * 6|;
```

答案:

```
    因此时针与分针的夹角度数为 |m * 30 + n * 0.5 - n * 6|;
```

详细资料参考:
《面试智力题 — 时针与分针夹角度数问题》

2. 用3升，5升杯子怎么量出4升水?

```
    (1) 将 5 升杯子装满水，然后倒入 3 升杯子中，之后 5 升杯子还剩 2 升水。

    (2) 将 3 升杯子的水倒出，然后将 5 升杯子中的 2 升水倒入 3 升杯子中。

    (3) 将 5 升杯子装满水，然后向 3 升杯子中倒水，直到 3 升杯子装满为止，此时 5 升杯子中就还剩 4 升水。
```

3. 四个药罐中有一个浑浊的药罐，浑浊的每片药片都比其他三个干净的药罐多一克，如何只用一次天平找出浑浊的药罐?

```
    由于浑浊的每片药片比正常药片都多出了一克，因此我认为可以通过控制药片的数量来实现判断。

    (1) 首先将每个药罐进行编号，分别标记为 1、2、3、4 号药罐。

    (2) 然后从 1 号药罐中取出 1 片药片，从 2 号药罐中取出 2 片药片，从 3 号药罐中取出 3 片药片，从 4 号药罐中取出 4
        片药片。

    (3) 将 10 片药片使用天平称重，药片的重量比正常重量多出几克，就是哪一号药罐的问题。
```

4. 四张卡片，卡片正面是数字，反面是字母。现在桌上四张卡片，状态为 a 1 b 2 现在我想要证明 a 的反面必然是 1 ，我只能翻两张牌，我翻哪两张?

```
    我认为证明 a 的反面一定是 1 的充要条件为 a 的反面为 1，并且 2 的反面不能为 a，因此应该翻 a 和 2 两张牌。
```

5. 赛马问题，25 匹马，5 个赛道，最少几次能选出最快的三匹马?

```
    我认为一共至少需要 7 次才能选出最快的三匹马。

    （1）首先，我们将 25 匹马分为 5 组，每组进行比赛，选出每组最快的三匹马，其余的马由于已经不可能成为前三了，因此可以直
        接淘汰掉，那么我们现在还剩下了 15 匹马。

    （2）然后我们将 5 组中的第一名来进行一轮比赛，最终的结果能够确定最快的马一定是第一名，四五名的马以及它们对应组的其余
        马就可以淘汰掉了，因为它们已经没有进入前三的机会了。并且第二名那一组的第三名和第三组的第二第三名都可以淘汰掉了，
        它们也没有进入前三的机会了。因此我们最终剩下了第一名那一组的二三名和第二名那一组的一二名，以及第三名一共 5 匹马，
        它们都有竞争最快第二第三的机会。

    （3）最后一次对最后的 5 匹马进行比赛，选择最快的一二名作为最终结果的二三名，因此就能够通过 7 次比较，选择出最快的马。
```

6. 五队夫妇参加聚会，每个人不能和自己的配偶握手，只能最多和他人握手一次。A问了其他人，发现每个人的握手次数都不同，那么A的配偶握手了几次?

```
    （1）由于每个人不能和自己的配偶握手，并且最多只能和他人握手一次，因此一个人最多能握 8 次手。

    （2）因为 A 问了除自己配偶的其他人，每个人的握手次数都不同。因此一共有九种握手的情况，由于一个人最多只能握 8 次手，因
        此握手的情况分别为 0、1、2、3、4、5、6、7、8 这九种情况。

    （3）我们首先分析握了 8 次手的人，由于他和除了自己配偶的每一个人都握了一次手，因此其他人的握手次数都不为 0，因此只有
        他的配偶握手次数为0，由此我们可以知道握手次数为 8 的人和握手次数为 0 的人是配偶。

    （4）我们再来分析握了 7 次手的人，他和除了握了 0 次手以外的人都握了一次手，由于握了 8 次手的人和其余人也都握了一次手
        ，因此其他人的握手次数至少为 2 ，因此只有他的配偶的握手次数才能为 1。由此我们可以知道握手次数为 7 的人和握手次数
        为 1 的人是配偶。

    （5）依次可以类推，握手次数为 6 的人和握手次数为 2 的人为配偶，握手次数为 5 的人和握手次数为 3 的人为配偶。

    （6）最终剩下了握手次数为 4 的人，按照规律我们可以得知他的配偶的握手次数也为4。

    （7）由于 A 和其他人的握手次数都不同，因此我们可以得知握手次数为 4 的人就是 A。因此他的配偶的握手次数为 4 。
```

7. 你只能带行走 60 公里的油，只能在起始点加油，如何穿过 80 公里的沙漠?

```
    （1）先走到离起点 20 公里的地方，然后放下 20 公里的油在这，然后返回起点加油。

    （2）当第二次到达这时，车还剩 40 公里的油，加上上一次放在这的 20 公里的油，一共就有 60 公里的油，能够走完剩下的路
        程。
```

8. 烧一根不均匀的绳要用一个小时，如何用它来判断一个小时十五分钟?

```
    一共需要三根绳子，假设分别为 1、2、3 号绳子，每个绳子一共有 A、B 两端。

    （1）首先点燃 1 号绳子的 A、B 两端，然后点燃 2 号绳子的 A 端。

    （2）当 1 号绳子燃尽时，此时过去了半小时，然后同时点燃 2 号绳子的 B 端。

    （3）当 2 号绳子燃尽时，此时又过去了 15 分钟，然后同时点燃 3 号绳子的 A、B 两端。

    （4）当 3 号绳子燃尽时，又过去了半小时，以此一共加起来过去了一个小时十五分钟。
```

9. 有7克、2克砝码各一个，天平一只，如何只用这些物品三次将140克的盐分成50、90克各一份?

```
    （1） 第一次用 7 克砝码和 2 克砝码称取 9 克盐。

    （2） 第二次再用第一次称取的盐和砝码称取 16 克盐。

    （3） 第三次再用前两次称取的盐和砝码称取 25 克盐，这样就总共称取了 50 克盐，剩下的就是 90 克。
```

10. 有一辆火车以每小时15公里的速度离开洛杉矶直奔纽约，另一辆火车以第 小时20公里的速度从纽约开往洛杉矶。如果有一只鸟，以外30公里每小时的速度和 两辆火车现时启动，从洛杉矶出发，碰到另辆车后返回，依次在两辆火车来回的飞行，直道两面辆火车相遇，请问，这只小鸟飞行了多长距离?

```
    由于小鸟一直都在飞，直到两车相遇时才停下来。因此小鸟飞行的时间为两车相遇的时间，由于两车是相向而行，因此
    两车相遇的时间为总路程除以两车的速度之和，然后再用飞行的时间去乘以小鸟的速度，就能够得出小鸟飞行的距离。
```

11. 你有两个罐子，50个红色弹球，50个蓝色弹球，随机选出一个罐子，随机选取出一个弹球放入罐子，怎么给红色弹球最大的选中机会？在你的计划中，得到红球的准确几率是多少?

```
    第一个罐子里放一个红球，第二个罐子里放剩余的球，这样概率接近75%，这是概率最大的方法
```

12. 假设你有8个球，其中一个略微重一些，但是找出这个球的惟一方法是将两个球放在天平上对比。最少要称多少次才能找出这个较重的球?

```
    最少两次可以称出。

    首先将 8 个球分为 3 组，其中两组为 3 个球，一组为 2 个球。

    第一次将两组三个的球进行比较，如果两边相等，则说明重的球在最后一组里。第二次将最后一组的球进行比较即可。如
    果两边不等，则说明重的球在较重的一边，第二次只需从这一组中随机取两球出来比较即可判断。
```

13. 在房里有三盏灯，房外有三个开关，在房外看不见房内的情况，你只能进门一次，你用什么方法来区分那个开关控制那一盏灯?

```
    （1）首先打开一盏灯 10 分钟，然后打开第二盏。
    （2）进入房间，看看那盏灯亮，摸摸那盏灯热，热的是第一个开关打开的，亮的是第二个开关打开的，而剩下的就是第三个开关打开
        的。
```

14. 他们都各自买了两对黑袜和两对白袜，八对袜子的布质、大小完全相同，而每对袜子都有一张商标纸连着。两位盲人不小心将八对袜子混在一起。他们每人怎样才能取回黑袜和白袜各两对呢?

```
    将每一对袜子分开，一人拿一只袜子，因为袜子不分左右脚的，因此最后每个人都能取回白袜和黑袜两对。
```

15. 有三筐水果，一筐装的全是苹果，第二筐装的全是橘子，第三筐是橘子与苹果混在一起。筐上的标签都是骗人的，（就是说筐上的标签都是错的）你的任务是拿出其中一筐，从里面只拿一只水果，然后正确写出三筐水果的标签。

```
    从混合标签里取出一个水果，取出的是什么水果，就写上相应的标签。

    对应水果标签的筐的标签改为另一种水果。

    另一种水果标签的框改为混合。
```

16. 一个班级60%喜欢足球，70%喜欢篮球，80%喜欢排球，问即三种球都喜欢占比有多少?

```
    （1）首先确定最多的一种情况，就是 60% 喜欢足球的人同时也喜欢篮球和排球，此时为三种球都喜欢的人的最大比例。

    （2）然后确定最小的一种情况，根据题目可以知道有 40%的人不喜欢足球，30%的人不喜欢篮球，20%的人不喜欢排球，因此有最多
        90% 的人三种球中有一种球不喜欢，因此三种球都喜欢的人的最小比例为 10%。

    因此三种球都喜欢的人占比为 10%-60%
```

17. 五只鸡五天能下五个蛋，一百天下一百个蛋需要多少只鸡?

```
    五只鸡五天能下五个蛋，平均下来五只鸡每天能下一个蛋，因此五只鸡一百天就能够下一百个蛋。
```

更多的智力题可以参考：
《经典面试智力题200+题和解答》

剑指 offer 思路总结

本部分主要是笔者在练习剑指 offer 时所做的笔记，如果出现错误，希望大家指出！

题目

1. 二维数组中的查找

```
    题目：
    在一个二维数组中，每一行都按照从左到右递增的顺序排序，每一列都按照从上到下递增的顺序排序。请完成一个函数，输入这样的
    一个二维数组和一个整数，判断数组中是否含有该整数。


    思路：

    （1）第一种方式是使用两层循环依次遍历，判断是否含有该整数。这一种方式最坏情况下的时间复杂度为 O(n^2)。

    （2）第二种方式是利用递增序列的特点，我们可以从二维数组的右上角开始遍历。如果当前数值比所求的数要小，则将位置向下移动
        ，再进行判断。如果当前数值比所求的数要大，则将位置向左移动，再进行判断。这一种方式最坏情况下的时间复杂度为 O(n)。
```

2. 替换空格

```
1  题目：
2
3  请实现一个函数，将一个字符串中的空格替换成“%20”。例如，当字符串为 We Are Happy.则经过替换之后的字符串为 We%20
4  Are%20Happy
5
6
7  思路：
8
9  使用正则表达式，结合字符串的 replace 方法将空格替换为 “%20”
10
11 str.replace(/\s/g,"%20")
```

3. 从尾到头打印链表

```
1  题目：
2
3  输入一个链表，从尾到头打印链表每个节点的值。
4
5
6  思路：
7
8  利用栈来实现，首先根据头结点以此遍历链表节点，将节点加入到栈中。当遍历完成后，再将栈中元素弹出并打印，以此来实现。栈的
9  实现可以利用 Array 的 push 和 pop 方法来模拟。
```

4. 重建二叉树

```
1  题目：
2
3  输入某二叉树的前序遍历和中序遍历的结果，请重建出该二叉树。假设输入的前序遍历和中序遍历的结果中都不含重复的数字。例如输
4  入前序遍历序列 {1,2,4,7,3,5,6,8} 和中序遍历序列 {4,7,2,1,5,3,8,6}，则重建二叉树并返回。
5
6
7  思路：
8
9  利用递归的思想来求解，首先先序序列中的第一个元素一定是根元素。然后我们去中序遍历中寻找到该元素的位置，找到后该元素的左
10 边部分就是根节点的左子树，右边部分就是根节点的右子树。因此我们可以分别截取对应的部分进行子树的递归构建。使用这种方式的
11 时间复杂度为 O(n)，空间复杂度为 O(logn)。
```

5. 用两个栈实现队列

```
题目：

用两个栈来实现一个队列，完成队列的 Push 和 Pop 操作。


思路：

队列的一个基本特点是，元素先进先出。通过两个栈来模拟时，首先我们将两个栈分为栈1和栈2。当执行队列的 push 操作时，直接
将元素 push 进栈1中。当队列执行 pop 操作时，首先判断栈2是否为空，如果不为空则直接pop 元素。如果栈2为空，则将栈1中
的所有元素 pop 然后 push 到栈2中，然后再执行栈2的 pop 操作。

扩展：

当使用两个长度不同的栈来模拟队列时，队列的最大长度为较短栈的长度的两倍。
```

6. 旋转数组的最小数字

```
题目：

把一个数组最开始的若干个元素搬到数组的末尾，我们称之为数组的旋转。 输入一个非递减排序的数组的一个旋转，输出旋转数组的
最小元素。 例如数组{3,4,5,1,2}为{1,2,3,4,5}的一个旋转，该数组的最小值为1。 NOTE：给出的所有元素都大于0，若数组大
小为0，请返回0。


思路：

（1）我们输入的是一个非递减排序的数组的一个旋转，因此原始数组的值递增或者有重复。旋转之后原始数组的值一定和一个值相
        邻，并且不满足递增关系。因此我们就可以进行遍历，找到不满足递增关系的一对值，后一个值就是旋转数组的最小数字。

（2）二分法
```

相关资料可以参考：
[《旋转数组的最小数字》]

7. 斐波那契数列

题目：

大家都知道斐波那契数列，现在要求输入一个整数 n，请你输出斐波那契数列的第 n 项。 n<=39

思路：

斐波那契数列的规律是，第一项为0，第二项为1，第三项以后的值都等于前面两项的和，因此我们可以通过循环的方式，不断通过叠
加来实现第 n 项值的构建。通过循环而不是递归的方式来实现，时间复杂度降为了 O(n)，空间复杂度为 O(1)。

8. 跳台阶

题目：

一只青蛙一次可以跳上1级台阶，也可以跳上2级。求该青蛙跳上一个 n 级的台阶总共有多少种跳法。

思路：

跳台阶的问题是一个动态规划的问题，由于一次只能够跳1级或者2级，因此跳上 n 级台阶一共有两种方案，一种是从 n-1 跳上，一
种是从 n-2 级跳上，因此 f(n) = f(n-1) + f(n-2)。

和斐波那契数列类似，不过初始两项的值变为了 1 和 2，后面每项的值等于前面两项的和。

9. 变态跳台阶

题目：

一只青蛙一次可以跳上1级台阶，也可以跳上2级……它也可以跳上 n 级。求该青蛙跳上一个 n 级的台阶总共有多少种跳法。

思路：

变态跳台阶的问题同上一个问题的思考方案是一样的，我们可以得到一个结论是，每一项的值都等于前面所有项的值的和。

```
f(1) = 1
f(2) = f(2-1) + f(2-2)          //f(2-2) 表示2阶一次跳2阶的次数。
f(3) = f(3-1) + f(3-2) + f(3-3)
...
f(n) = f(n-1) + f(n-2) + f(n-3) + ... + f(n-(n-1)) + f(n-n)
```

再次总结可得

```
                | 1       ,(n=0 )
   f(n) =       | 1       ,(n=1 )
                | 2*f(n-1),(n>=2)
```

10. 矩形覆盖

```
    题目：

    我们可以用 2*1 的小矩形横着或者竖着去覆盖更大的矩形。请问用 n 个 2*1 的小矩形无重叠地覆盖一个 2*n 的大矩形，总共
    有多少种方法?


    思路：

    依旧是斐波那契数列的应用
```

11. 二进制中1的个数

```
    题目：

    输入一个整数，输出该数二进制表示中1的个数。其中负数用补码表示。


    思路：

    一个不为 0 的整数的二进制表示，一定会有一位为1。我们找到最右边的一位1，当我们将整数减去1时，最右边的一位1变为0，它后
    面的所有位都取反，因此将减一后的值与原值相与，我们就会能够消除最右边的一位1。因此判断一个二进制中1的个数，我们可以判
    断这个数可以经历多少次这样的过程。

    如：1100&1011=1000
```

12. 数值的整数次方

```
    题目：

    给定一个 double 类型的浮点数 base 和 int 类型的整数 exponent。求 base 的 exponent 次方。


    思路：

    首先我们需要判断 exponent 正负和零取值三种情况，根据不同的情况通过递归来实现。
```

13. 调整数组顺序使奇数位于偶数前面

题目：

输入一个整数数组，实现一个函数来调整该数组中数字的顺序，使得所有的奇数位于数组的前半部分，所有的偶数位于位于数组的后半
部分，并保证奇数和奇数，偶数和偶数之间的相对位置不变。

思路：

由于需要考虑到调整之后的稳定性，因此我们可以使用辅助数组的方式。首先对数组中的元素进行遍历，每遇到一个奇数就将它加入到
奇数辅助数组中，每遇到一个偶数，就将它将入到偶数辅助数组中。最后再将两个数组合并。这一种方法的时间复杂度为 O(n)，空间
复杂度为 O(n)。

14. 链表中倒数第 k 个节点

题目：

输入一个链表，输出该链表中倒数第 k 个结点。

思路：

使用两个指针，先让第一个和第二个指针都指向头结点，然后再让第二个指针走 k-1 步，到达第 k 个节点。然后两个指针同时向后
移动，当第二个指针到达末尾时，第一个指针指向的就是倒数第 k 个节点了。

15. 反转链表

题目：

输入一个链表，反转链表后，输出链表的所有元素。

思路：

通过设置三个变量 pre、current 和 next，分别用来保存前继节点、当前节点和后继结点。从第一个节点开始向后遍历，首先将当
前节点的后继节点保存到 next 中，然后将当前节点的后继节点设置为 pre，然后再将 pre 设置为当前节点，current 设置为 ne
xt 节点，实现下一次循环。

16. 合并两个排序的链表

题目：

输入两个单调递增的链表，输出两个链表合成后的链表，当然我们需要合成后的链表满足单调不减规则。

思路：

通过递归的方式，依次将两个链表的元素递归进行对比。

17. 树的子结构

题目：

输入两棵二叉树A、B，判断 B 是不是 A 的子结构。（ps：我们约定空树不是任意一个树的子结构）

思路：

通过递归的思想来解决

第一步首先从树 A 的根节点开始遍历，在左右子树中找到和树 B 根结点的值一样的结点 R 。
第二步两棵树同时从 R 节点和根节点以相同的遍历方式进行遍历，依次比较对应的值是否相同，当树 B 遍历结束时，结束比较。

18. 二叉树的镜像

题目：

操作给定的二叉树，将其变换为源二叉树的镜像。

思路：

从根节点开始遍历，首先通过临时变量保存左子树的引用，然后将根节点的左右子树的引用交换。然后再递归左右节点的子树交换。

19. 顺时针打印矩阵

题目：

输入一个矩阵，按照从外向里以顺时针的顺序依次打印出每一个数字，
例如，如果输入如下矩阵：
```
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 16
```
则依次打印出数字1,2,3,4,8,12,16,15,14,13,9,5,6,7,11,10

思路：

（1）根据左上角和右下角可以定位出一次要旋转打印的数据。一次旋转打印结束后，往对角分别前进和后退一个单位，可以确定下一次需要打印的数据范围。

（2）使用模拟魔方逆时针解法，每打印一行，则将矩阵逆时针旋转 90 度，打印下一行，依次重复。

20. 定义一个栈，实现 min 函数

题目：

定义栈的数据结构，请在该类型中实现一个能够得到栈最小元素的 min 函数。

思路：

使用一个辅助栈，每次将数据压入数据栈时，就把当前栈里面最小的值压入辅助栈当中。这样辅助栈的栈顶数据一直是数据栈中最小的值。

21. 栈的压入弹出

题目：

输入两个整数序列，第一个序列表示栈的压入顺序，请判断第二个序列是否为该栈的弹出顺序。假设压入栈的所有数字均不相等。例如序列 1,2,3,4,5 是某栈的压入顺序，序列 4,5,3,2,1 是该压栈序列对应的一个弹出序列，但 4,3,5,1,2 就不可能是该压栈序列的弹出序列。（注意：这两个序列的长度是相等的）

思路：

我们可以使用一个辅助栈的方式来实现，首先遍历压栈顺序，依次将元素压入辅助栈中，每次压入元素后我们首先判断该元素是否与出栈顺序中的此刻位置的元素相等，如果不相等，则将元素继续压栈，如果相等，则将辅助栈中的栈顶元素出栈，出栈后，将出栈顺序中的位置后移一位继续比较。当压栈顺序遍历完成后，如果辅助栈不为空，则说明该出栈顺序不正确。

22. 从上往下打印二叉树

```
    题目：

    从上往下打印出二叉树的每个节点，同层节点从左至右打印。


    思路：

    本质上是二叉树的层序遍历，可以通过队列来实现。首先将根节点入队。然后对队列进行出队操作，每次出队时，将出队元素的左右子
    节点依次加入到队列中，直到队列长度变为 0 时，结束遍历。
```

23. 二叉搜索树的后序遍历

```
    题目：

    输入一个整数数组，判断该数组是不是某二叉搜索树的后序遍历的结果。如果是则输出 Yes，否则输出 No。假设输入的数组的任意两
    个数字都互不相同。


    思路：

    对于一个合法而二叉树的后序遍历来说，最末尾的元素为根元素。该元素前面的元素可以划分为两个部分，一部分为该元素的左子树，
    所有元素的值比根元素小，一部分为该元素的右子树，所有的元素的值比该根元素大。并且每一部分都是一个合法的后序序列，因此我
    们可以利用这些特点来递归判断。
```

24. 二叉树中和为某一值路径

```
    题目：

    输入一颗二叉树和一个整数，打印出二叉树中结点值的和为输入整数的所有路径。路径定义为从树的根结点开始往下一直到叶结点所经
    过的结点形成一条路径。


    思路：

    通过对树进行深度优先遍历，遍历时保存当前节点的值并判断是否和期望值相等，如果遍历到叶节点不符合要求则回退处理。
```

25. 复杂链表的复制

题目：

输入一个复杂链表（每个节点中有节点值，以及两个指针，一个指向下一个节点，另一个特殊指针指向任意一个节点），返回结果为
复制后复杂链表的 head。（注意，输出结果中请不要返回参数中的节点引用，否则判题程序会直接返回空）

思路：

（1）第一种方式，首先对原有链表每个节点进行复制，通过 next 连接起来。然后当链表复制完成之后，再来设置每个节点的 ra
ndom 指针，这个时候每个节点的 random 的设置都需要从头结点开始遍历，因此时间的复杂度为 O(n^2)。

（2）第二种方式，首先对原有链表每个节点进行复制，并且使用 Map 以键值对的方式将原有节点和复制节点保存下来。当链表复
制完成之后，再来设置每个节点的 random 指针，这个时候我们通过 Map 中的键值关系就可以获取到对应的复制节点，因此
不必再从头结点遍历，将时间的复杂度降低为了 O(n)，但是空间复杂度变为了 O(n)。这是一种以空间换时间的做法。

（3）第三种方式，首先对原有链表的每个节点进行复制，并将复制后的节点加入到原有节点的后面。当链表复制完成之后，再进行
random 指针的设置，由于每个节点后面都跟着自己的复制节点，因此我们可以很容易的获取到 random 指向对应的复制节点
。最后再将链表分离，通过这种方法我们也能够将时间复杂度降低为 O(n)。

26. 二叉搜索树与双向链表

题目：

输入一棵二叉搜索树，将该二叉搜索树转换成一个排序的双向链表。要求不能创建任何新的结点，只能调整树中结点指针的指向。

思路：

需要生成一个排序的双向列表，那么我们应该通过中序遍历的方式来调整树结构，因为只有中序遍历，返回才是一个从小到大的排序
序列。

基本的思路是我们首先从根节点开始遍历，先将左子树调整为一个双向链表，并将左子树双向链表的末尾元素的指针指向根节点，并
将根节点的左节点指向末尾节点。再将右子树调整为一个双向链表，并将右子树双向链表的首部元素的指针指向根元素，再将根节点
的右节点指向首部节点。通过对左右子树递归调整，因此来实现排序的双向链表的构建。

27. 字符串的排列

题目：

输入一个字符串，按字典序打印出该字符串中字符的所有排列。例如输入字符串 abc，则打印出由字符 a,b,c 所能排列出来的所有
字符串 abc,acb,bac,bca,cab 和 cba。输入描述：输入一个字符串，长度不超过9（可能有字符重复），字符只包括大小写字母。

思路：

我们可以把一个字符串看做是两个部分，第一部分为它的第一个字符，第二部分是它后面的所有字符。求整个字符串的一个全排列，可
以看做两步，第一步是求所有可能出现在第一个位置的字符，即把第一个字符和后面的所有字符交换。第二步就是求后面所有字符的一
个全排列。因此通过这种方式，我们可以以递归的思路来求出当前字符串的全排列。

详细资料可以参考：
《字符串的排列》

28. 数组中出现次数超过一半的数字

题目：

数组中有一个数字出现的次数超过数组长度的一半。请找出这个数字。例如输入一个长度为9的数组{1,2,3,2,2,2,5,4,2}。由于数
字2在数组中出现了5次，超过数组长度的一半，因此输出2。如果不存在则输出0。

思路：

（1）对数组进行排序，排序后的中位数就是所求数字。这种方法的时间复杂度取决于我们采用的排序方法的时间复杂度，因此最快为
O(nlogn)。

（2）由于所求数字的数量超过了数组长度的一半，因此排序后的中位数就是所求数字。因此我们可以将问题简化为求一个数组的中
位数问题。其实数组并不需要全排序，只需要部分排序。我们通过利用快排中的 partition 函数来实现，我们现在数组中随
机选取一个数字，而后通过 partition 函数返回该数字在数组中的索引 index，如果 index 刚好等于 n/2，则这个数字
便是数组的中位数，也即是要求的数，如果 index 大于 n/2，则中位数肯定在 index 的左边，在左边继续寻找即可，反之
在右边寻找。这样可以只在 index 的一边寻找，而不用两边都排序，减少了一半排序时间，这种方法的时间复杂度为 O(n)。

（3）由于该数字的出现次数比所有其他数字出现次数的和还要多，因此可以考虑在遍历数组时保存两个值：一个是数组中的一个数
字，一个是次数。当遍历到下一个数字时，如果下一个数字与之前保存的数字相同，则次数加1，如果不同，则次数减1，如果
次数为0，则需要保存下一个数字，并把次数设定为1。由于我们要找的数字出现的次数比其他所有数字的出现次数之和还要大，
则要找的数字肯定是最后一次把次数设为1时对应的数字。该方法的时间复杂度为O(n)，空间复杂度为 O(1)。

详细资料可以参考：
《出现次数超过一半的数字》

29. 最小的 K 个数

题目：

输入 n 个整数，找出其中最小的 K 个数。例如输入 4,5,1,6,2,7,3,8 这8个数字，则最小的4个数字是 1,2,3,4 。

思路：

（1）第一种思路是首先将数组排序，排序后再取最小的 k 个数。这一种方法的时间复杂度取决于我们选择的排序算法的时间复杂
度，最好的情况下为 O(nlogn)。

（2）第二种思路是由于我们只需要获得最小的 k 个数，这 k 个数不一定是按序排序的。因此我们可以使用快速排序中的 part
ition函数来实现。每一次选择一个枢纽值，将数组分为比枢纽值大和比枢纽值小的两个部分，判断枢纽值的位置，如果该枢
纽值的位置为 k-1 的话，那么枢纽值和它前面的所有数字就是最小的 k 个数。如果枢纽值的位置小于 k-1 的话，假设枢
纽值的位置为 n-1，那么我们已经找到了前 n 小的数字了，我们就还需要到后半部分去寻找后半部分 k-n 小的值，进行划

分。当该枢纽值的位置比 k-1大时，说明最小的 k 个值还在左半部分，我们需要继续对左半部分进行划分。这一种方法的平

均时间复杂度为 O(n)。

（3）第三种方法是维护一个容量为 k 的最大堆。对数组进行遍历时，如果堆的容量还没有达到 k ，则直接将元素加入到堆中，这

就相当于我们假设前 k 个数就是最小的 k 个数。对 k 以后的元素遍历时，我们将该元素与堆的最大值进行比较，如果比最

大值小，那么我们则将最大值与其交换，然后调整堆。如果大于等于堆的最大值，则继续向后遍历，直到数组遍历完成。这一

种方法的平均时间复杂度为 O(nlogk)。

详细资料可以参考：
《寻找最小的 k 个数》

30. 连续子数组的最大和

题目：

HZ 偶尔会拿些专业问题来忽悠那些非计算机专业的同学。今天测试组开完会后，他又发话了:在古老的一维模式识别中，常常需要计

算连续子向量的最大和,当向量全为正数的时候，问题很好解决。但是，如果向量中包含负数，是否应该包含某个负数，并期望旁边的

正数会弥补它呢？例如：{6,-3,-2,7,-15,1,2,2}，连续子向量的最大和为8（从第0个开始，到第3个为止）。你会不会被他忽悠

住？（子向量的长度至少是1）

思路：

（1）第一种思路是直接暴力求解的方式，先以第一个数字为首往后开始叠加，叠加的过程中保存最大的值。然后再以第二个数字为首

往后开始叠加，并与先前保存的最大的值进行比较。这一种方法的时间复杂度为 O(n^2)。

（2）第二种思路是，首先我们观察一个最大和的连续数组的规律，我们可以发现，子数组一定是以正数开头的，中间包含了正负数。

因此我们可以从第一个数开始向后叠加，每次保存最大的值。叠加的值如果为负数，则将叠加值初始化为0，因为后面的数加上负

数只会更小，因此需要寻找下一个正数开始下一个子数组的判断。一直往后判断，直到这个数组遍历完成为止，得到最大的值。

使用这一种方法的时间复杂度为 O(n)。

详细资料可以参考：
《连续子数组的最大和》

31. 整数中1出现的次数（待深入理解）

```
    题目：

    求出1~13的整数中1出现的次数，并算出100~1300的整数中1出现的次数？为此他特别数了一下1~13中包含1的数字有1、10、11、
    12、13因此共出现6次，但是对于后面问题他就没辙了。ACMer希望你们帮帮他，并把问题更加普遍化，可以很快的求出任意非负整
    数区间中1出现的次数。

    思路：

     （1）第一种思路是直接遍历每个数，然后将判断每个数中 1 的个数，一直叠加。

     （2）第二种思路是求出1出现在每位上的次数，然后进行叠加。
```

详细资料可以参考：
《从1到n整数中1出现的次数：O(logn)算法》

32. 把数组排成最小的数

```
    题目：

    输入一个正整数数组，把数组里所有数字拼接起来排成一个数，打印能拼接出的所有数字中最小的一个。例如输入数组{3，32，321
    }，则打印出这三个数字能排成的最小数字为321323。


    思路：

     （1）求出数组的全排列，然后对每个排列结果进行比较。

     （2）利用排序算法实现，但是比较时，比较的并不是两个元素的大小，而是两个元素正序拼接和逆序拼接的大小，如果逆序拼接的
          结果更小，则交换两个元素的位置。排序结束后，数组的顺序则为最小数的排列组合顺序。
```

详细资料可以参考：
《把数组排成最小的数》

33. 丑数（待深入理解）

题目：

把只包含质因子2、3和5的数称作丑数。例如6、8都是丑数，但14不是，因为它包含因子7。 习惯上我们把1当做是第一个丑数。求
按从小到大的顺序的第 N 个丑数。

思路：

（1）判断一个数是否为丑数，可以判断该数不断除以2，最后余数是否为1。判断该数不断除以3，最后余数是否为1。判断不断除以
5，最后余数是否为1。在不考虑时间复杂度的情况下，可以依次遍历找到第 N 个丑数。

（2）使用一个数组来保存已排序好的丑数，后面的丑数由前面生成。

34. 第一个只出现一次的字符

题目：

在一个字符串（1<=字符串长度<=10000，全部由大写字母组成）中找到第一个只出现一次的字符，并返回它的位置。

思路：

（1）第一种思路是，从前往后遍历每一个字符。每遍历一个字符，则将字符与后边的所有字符依次比较，判断是否含有相同字符。这
一种方法的时间复杂度为 O(n^2)。

（2）第二种思路是，首先对字符串进行一次遍历，将字符和字符出现的次数以键值对的形式存储在 Map 结构中。然后第二次遍历时
，去 Map 中获取对应字符出现的次数，找到第一个只出现一次的字符。这一种方法的时间复杂度为 O(n)。

35. 数组中的逆序对

题目：

在数组中的两个数字，如果前面一个数字大于后面的数字，则这两个数字组成一个逆序对。输入一个数组，求出这个数组中的逆序对
的总数 P。

思路：

（1）第一种思路是直接求解的方式，顺序扫描整个数组。每扫描到一个数字的时候，逐个比较该数字和它后面的数字的大小。如果

后面的数字比它小，则这两个数字就组成了一个逆序对。假设数组中含有 n 个数字。由于每个数字都要和 O(n) 个数字作比较，因此这个算法的时间复杂度是 O(n^2)。

（2）第二种方式是使用归并排序的方式，通过利用归并排序分解后进行合并排序时，来进行逆序对的统计，这一种方法的时间复杂度为 O(nlogn)。

详细资料可以参考：
《数组中的逆序对》

36. 两个链表的第一个公共结点

题目：

输入两个链表，找出它们的第一个公共结点。

思路：

（1）第一种方法是在第一个链表上顺序遍历每个结点，每遍历到一个结点的时候，在第二个链表上顺序遍历每个结点。如果在第二个链表上有一个结点和第一个链表上的结点一样，说明两个链表在这个结点上重合，于是就找到了它们的公共结点。如果第一个链表的长度为 m，第二个链表的长度为 n。这一种方法的时间复杂度是 O(mn) 。

（2）第二种方式是利用栈的方式，通过观察我们可以发现两个链表的公共节点，都位于链表的尾部，以此我们可以分别使用两个栈，依次将链表元素入栈。然后在两个栈同时将元素出栈，比较出栈的节点，最后一个相同的节点就是我们要找的公共节点。这一种方法的时间复杂度为 O(m+n)，空间复杂度为 O(m+n)。

（3）第三种方式是，首先分别遍历两个链表，得到两个链表的长度。然后得到较长的链表与较短的链表长度的差值。我们使用两个指针来分别对两个链表进行遍历，首先将较长链表的指针移动 n 步，n 为两个链表长度的差值，然后两个指针再同时移动，判断所指向节点是否为同一节点。这一种方法的时间复杂度为 O(m+n)，相同对于上一种方法不需要额外的空间。

详细资料可以参考：
《两个链表的第一个公共结点》

37. 数字在排序数组中出现的次数

题目：

统计一个数字：在排序数组中出现的次数。例如输入排序数组 { 1, 2, 3, 3, 3, 3, 4, 5} 和数字 3 ，由于 3 在这个数组中出现了 4 次，因此输出 4 。

思路：

（1）第一种方法是直接对数组顺序遍历的方式，通过这种方法来统计数字的出现次数。这种方法的时间复杂度为 O(n)。

（2）第二种方法是使用二分查找的方法，由于数组是排序好的数组，因此相同数字是排列在一起的。统计数字出现的次数，我们需要
去找到该段数字开始和结束的位置，以此来确定数字出现的次数。因此我们可以使用二分查找的方式来确定该数字的开始和结束
位置。如果我们第一次我们数组的中间值为 k ，如果 k 值比所求值大的话，那么我们下一次只需要判断前面一部分就行了，如
果 k值比所求值小的话，那么我们下一次就只需要判断后面一部分就行了。如果 k 值等于所求值的时候，我们则需要判断该值
是否为开始位置或者结束位置。如果是开始位置，那么我们下一次需要到后半部分去寻找结束位置。如果是结束位置，那么我们
下一次需要到前半部分去寻找开始位置。如果既不是开始位置也不是结束位置，那么我们就分别到前后两个部分去寻找开始和结
束位置。这一种方法的平均时间复杂度为 O(logn)。

38. 二叉树的深度

题目：

输入一棵二叉树，求该树的深度。从根结点到叶结点依次经过的结点（含根、叶结点）形成树的一条路径，最长路径的长度为树的深
度。

思路：

根节点的深度等于左右深度较大值加一，因此可以通过递归遍历来实现。

39. 平衡二叉树

题目：

输入一棵二叉树，判断该二叉树是否是平衡二叉树。

思路：

（1）在遍历树的每个结点的时候，调用函数得到它的左右子树的深度。如果每个结点的左右子树的深度相差都不超过 1 ，那么它就是一棵平衡的二叉树。使用这种方法时，节点会被多次遍历，因此会造成效率不高的问题。

（2）在求一个节点的深度时，同时判断它是否平衡。如果不平衡则直接返回 -1，否则返回树高度。如果一个节点的一个子树的深度为-1，那么就直接向上返回 -1 ，该树已经是不平衡的了。通过这种方式确保了节点只能够被访问一遍。

40. 数组中只出现一次的数字

题目：

一个整型数组里除了两个数字之外，其他的数字都出现了两次。请写程序找出这两个只出现一次的数字。

思路：

（1）第一种方式是依次遍历数组，记录下数字出现的次数，从而找出两个只出现一次的数字。

（2）第二种方式，根据位运算的异或的性质，我们可以知道两个相同的数字异或等于0，一个数和 0 异或还是它本身。由于数组中的其他数字都是成对出现的，因此我们可以将数组中的所有数依次进行异或运算。如果只有一个数出现一次的话，那么最后剩下的就是落单的数字。如果是两个数只出现了一次的话，那么最后剩下的就是这两个数异或的结果。这个结果中的1表示的是 A 和 B 不同的位。我们取异或结果的第一个1所在的位数，假如是第3位，接着通过比较第三位来将数组分为两组，相同数字一定会被分到同一组。分组完成后再按照依次异或的思路，求得剩余数字即为两个只出现一次的数字。

41. 和为 S 的连续正数序列

题目：

小明很喜欢数学，有一天他在做数学作业时，要求计算出9~16的和，他马上就写出了正确答案是100。但是他并不满足于此，他在想究

竟有多少种连续的正数序列的和为100（至少包括两个数）。没多久，他就得到另一组连续正数和为100的序列：18,19,20,21,22。

现在把问题交给你，你能不能也很快的找出所有和为 S 的连续正数序列? Good Luck!输出描述：输出所有和为S的连续正数序列。序

列内按照从小至大的顺序，序列间按照开始数字从小到大的顺序。

思路：

维护一个正数序列数组，数组中初始只含有值1和2，然后从3依次往后遍历，每遍历到一个元素则将这个元素加入到序列数组中，然后

判断此时序列数组的和。如果序列数组的和大于所求值，则将第一个元素（最小的元素弹出）。如果序列数组的和小于所求值，则继续

往后遍历，将元素加入到序列中继续判断。当序列数组的和等于所求值时，打印出此时的正数序列，然后继续往后遍历，寻找下一个连

续序列，直到数组遍历完成终止。

详细资料可以参考：
《和为 s 的连续正数序列》

42. 和为 S 的两个数字

题目：

输入一个递增排序的数组和一个数字 S，在数组中查找两个数，是的他们的和正好是 S，如果有多对数字的和等于 S，输出两个数

的乘积最小的。输出描述：对应每个测试案例，输出两个数，小的先输出。

思路：

首先我们通过规律可以发现，和相同的两个数字，两个数字的差值越大，乘积越小。因此我们只需要从数组的首尾开始找到第一对和

为 s 的数字对进行了。因此我们可以使用双指针的方式，左指针初始指向数组的第一个元素，右指针初始指向数组的最后一个元素

。然后首先判断两个指针指向的数字的和是否为 s ，如果为 s ，两个指针指向的数字就是我们需要寻找的数字对。如果两数的和

比 s 小，则将左指针向左移动一位后继续判断。如果两数的和比 s 大，则将右指针向右移动一位后继续判断。

详细资料可以参考：
《和为 S 的字符串》

43. 左旋转字符串

```
    题目：

    汇编语言中有一种移位指令叫做循环左移（ROL），现在有个简单的任务，就是用字符串模拟这个指令的运算结果。对于一个给定的
    字符序列 S，请你把其循环左移 K 位后的序列输出。例如，字符序列 S=”abcXYZdef”，要求输出循环左移3位后的结果，即 “X
    YZdefabc”。是不是很简单？OK，搞定它！


    思路：

    字符串裁剪后拼接
```

44. 翻转单词顺序列

```
    题目：

    牛客最近来了一个新员工 Fish，每天早晨总是会拿着一本英文杂志，写些句子在本子上。同事 Cat 对 Fish 写的内容颇感兴趣，有
    一天他向 Fish 借来翻看，但却读不懂它的意思。例如，“student. a am I”。后来才意识到，这家伙原来把句子单词的顺序翻转了
    ，正确的句子应该是“I am a student.”。Cat 对一一的翻转这些单词顺序可不在行，你能帮助他么？


    思路：

    通过空格将单词分隔，然后将数组反序后，重新拼接为字符串。
```

45. 扑克牌的顺子

```
    题目：

    LL 今天心情特别好，因为他去买了一副扑克牌，发现里面居然有2个大王，2个小王（一副牌原本是54张^_^）...他随机从中抽出
    了5张牌，想测测自己的手气，看看能不能抽到顺子，如果抽到的话，他决定去买体育彩票，嘿嘿！！“红心 A，黑桃3，小王，大王
    ，方片5”，“Oh My God!”不是顺子..... LL 不高兴了，他想了想，决定大\小王可以看成任何数字，并且 A 看作1，J 为11，
    Q 为12，K 为13。上面的5张牌就可以变成“1,2,3,4,5”（大小王分别看作2和4），“So Lucky!”。LL 决定去买体育彩票啦。
    现在，要求你使用这幅牌模拟上面的过程，然后告诉我们 LL 的运气如何。为了方便起见，你可以认为大小王是0。
```

```
    思路：

    首先判断5个数字是不是连续的，最直观的方法是把数组排序。值得注意的是，由于 0 可以当成任意数字，我们可以用 0 去补满数
    组中的空缺。如果排序之后的数组不是连续的，即相邻的两个数字相隔若干个数字，但只要我们有足够的。可以补满这两个数字的空
    缺，这个数组实际上还是连续的。

    于是我们需要做 3 件事情：首先把数组排序，再统计数组中 0 的个数，最后统计排序之后的数组中相邻数字之间的空缺总数。如
    果空缺的总数小于或者等于 0 的个数，那么这个数组就是连续的：反之则不连续。最后，我们还需要注意一点：如果数组中的非 0
    数字重复出现，则该数组不是连续的。换成扑克牌的描述方式就是如果一副牌里含有对子，则不可能是顺子。
```

详细资料可以参考：
《扑克牌的顺子》

46. 圆圈中最后剩下的数字（约瑟夫环问题）

```
    题目：

    0, 1, … , n-1 这 n 个数字排成一个圈圈，从数字 0 开始每次从圆圈里删除第 m 个数字。求出这个圈圈里剩下的最后一个数
    字。


    思路：

     （1）使用环形链表进行模拟。

     （2）根据规律得出（待深入理解）
```

详细资料可以参考：
《圆圈中最后剩下的数字》

47. 1+2+3+...+n

题目：

求 1+2+3+...+n，要求不能使用乘除法、for、while、if、else、switch、case 等关键字及条件判断语句（A?B:C）。

思路：

由于不能使用循环语句，因此我们可以通过递归来实现。并且由于不能够使用条件判断运算符，我们可以利用 && 操作符的短路特
性来实现。

48. 不用加减乘除做加法

题目：

写一个函数，求两个整数之和，要求在函数体内不得使用 +、-、×、÷ 四则运算符号。

思路：

通过位运算，递归来实现。

49. 把字符串转换成整数。

题目：

将一个字符串转换成一个整数，要求不能使用字符串转换整数的库函数。数值为0或者字符串不是一个合法的数值则返回 0。输入描
述：输入一个字符串，包括数字字母符号，可以为空。输出描述：如果是合法的数值表达则返回该数字，否则返回0。

思路：

首先需要进行符号判断，其次我们根据字符串的每位通过减0运算转换为整数和，依次根据位数叠加。

50. 数组中重复的数字

题目：

在一个长度为 n 的数组里的所有数字都在 0 到 n-1 的范围内。数组中某些数字是重复的，但不知道有几个数字重复了，也不知
道每个数字重复了几次。请找出数组中任意一个重复的数字。

```
    思路：

    （1）首先将数组排序，排序后再进行判断。这一种方法的时间复杂度为 O(nlogn)。

    （2）使用 Map 结构的方式，依次记录下每一个数字出现的次数，从而可以判断是否出现重复数字。这一种方法的时间复杂度为 O
        (n)，空间复杂度为 O(n)。

    （3）从数组首部开始遍历，每遍历一个数字，则将该数字和它的下标相比较，如果数字和下标不等，则将该数字和它对应下标的值
        交换。如果对应的下标值上已经是正确的值了，那么说明当前元素是一个重复数字。这一种方法相对于上一种方法来说不需要
        额外的内存空间。
```

51. 构建乘积数组

```
    题目：

    给定一个数组 A[0,1,...,n-1]，请构建一个数组 B[0,1,...,n-1]，其中 B 中的元素 B[i]=A[0]*A[1]*...*A[i-1]*A
    [i+1]*...*A[n-1]。不能使用除法。


    思路：

    （1）  C[i]=A[0]×A[1]×...×A[i-1]=C[i-1]×A[i-1]

           D[i]=A[i+1]×...×A[n-1]=D[i+1]×A[i+1]

           B[i]=C[i]×D[i]

           将乘积分为前后两个部分，分别循环求出后，再进行相乘。

    （2）上面的方法需要额外的内存空间，我们可以引入中间变量的方式，来降低空间复杂度。（待深入理解）
```

详细资料可以参考：
《构建乘积数组》

52. 正则表达式的匹配

```
    题目：

    请实现一个函数用来匹配包括'.'和'*'的正则表达式。模式中的字符'.'表示任意一个字符，而'*'表示它前面的字符可以出现任
    意次（包含0次）。 在本题中，匹配是指字符串的所有字符匹配整个模式。例如，字符串"aaa"与模式"a.a"和"ab*ac*a"匹配，
    但是与"aa.a"和"ab*a"均不匹配。


    思路：

     （1）状态机思路（待深入理解）
```

详细资料可以参考：
《正则表达式匹配》

53. 表示数值的字符串

```
    题目：

    请实现一个函数用来判断字符串是否表示数值（包括整数和小数）。例如，字符串"+100","5e2","-123","3.1416"和"-1E-
    16"都表示数值。 但是"12e","1a3.14","1.2.3","+-5"和"12e+4.3"都不是。、


    思路：

    利用正则表达式实现
```

54. 字符流中第一个不重复的字符

```
    题目：

    请实现一个函数用来找出字符流中第一个只出现一次的字符。例如，当从字符流中只读出前两个字符 "go" 时，第一个只出现一次
    的字符是 "g" 。当从该字符流中读出前六个字符 "google" 时，第一个只出现一次的字符是"l"。 输出描述：如果当前字符流
    没有存在出现一次的字符，返回#字符。


    思路：

    同第 34 题
```

55. 链表中环的入口结点

```
    题目：

    一个链表中包含环，如何找出环的入口结点?



    思路：

    首先使用快慢指针的方式我们可以判断链表中是否存在环，当快慢指针相遇时，说明链表中存在环。相遇点一定存在于环中，因此我
    们可以从使用一个指针从这个点开始向前移动，每移动一个点，环的长度加一，当指针再次回到这个点的时候，指针走了一圈，因此
    通过这个方法我们可以得到链表中的环的长度，我们将它记为 n 。

    然后我们设置两个指针，首先分别指向头结点，然后将一个指针先移动 n 步，然后两个指针再同时移动，当两个指针相遇时，相遇
    点就是环的入口节点。
```

详细资料可以参考：
《链表中环的入口结点》
《《剑指offer》——链表中环的入口结点》

56. 删除链表中重复的结点

```
    题目：

    在一个排序的链表中，存在重复的结点，请删除该链表中重复的结点，重复的结点不保留，返回链表头指针。例如，链表1->2->3-
    >3->4->4->5 处理后为 1->2->5



    思路：

    解决这个问题的第一步是确定删除的参数。当然这个函数需要输入待删除链表的头结点。头结点可能与后面的结点重复，也就是说头
    结点也可能被删除，所以在链表头额外添加一个结点。

    接下来我们从头遍历整个链表。如果当前结点的值与下一个结点的值相同，那么它们就是重复的结点，都可以被删除。为了保证删除
    之后的链表仍然是相连的而没有中间断开，我们要把当前的前一个结点和后面值比当前结点的值要大的结点相连。我们要确保 prev
    要始终与下一个没有重复的结点连接在一起。
```

57. 二叉树的下一个结点

题目：

给定一棵二叉树和其中的一个结点，如何找出中序遍历顺序的下一个结点？树中的结点除了有两个分别指向左右子结点的指针以外，
还有一个指向父节点的指针。

思路：

这个问题我们可以分为三种情况来讨论。

第一种情况，当前节点含有右子树，这种情况下，中序遍历的下一个节点为该节点右子树的最左子节点。因此我们只要从右子节点
出发，一直沿着左子节点的指针，就能找到下一个节点。

第二种情况是，当前节点不含有右子树，并且当前节点为父节点的左子节点，这种情况下中序遍历的下一个节点为当前节点的父节
点。

第三种情况是，当前节点不含有右子树，并且当前节点为父节点的右子节点，这种情况下我们沿着父节点一直向上查找，直到找到
一个节点，该节点为父节点的左子节点。这个左子节点的父节点就是中序遍历的下一个节点。

58. 对称二叉树

题目：

请实现一个函数来判断一棵二叉树是不是对称的。如果一棵二叉树和它的镜像一样，那么它是对称的。

思路：

我们对一颗二叉树进行前序遍历的时候，是先访问左子节点，然后再访问右子节点。因此我们可以定义一种对称的前序遍历的方式
，就是先访问右子节点，然后再访问左子节点。通过比较两种遍历方式最后的结果是否相同，以此来判断该二叉树是否为对称二叉
树。

59. 按之字形顺序打印二叉树（待深入理解）

```
    题目：

    请实现一个函数按照之字形顺序打印二叉树，即第一行按照从左到右的顺序打印，第二层按照从右到左的顺序打印，即第一行按照
    从左到右的顺序打印，第二层按照从右到左顺序打印，第三行再按照从左到右的顺序打印，其他以此类推。


    思路：

    按之字形顺序打印二叉树需要两个栈。我们在打印某一行结点时，把下一层的子结点保存到相应的栈里。如果当前打印的是奇数层
    ，则先保存左子结点再保存右子结点到一个栈里；如果当前打印的是偶数层，则先保存右子结点再保存左子结点到第二个栈里。每
    一个栈遍历完成后进入下一层循环。
```

详细资料可以参考：
[《按之字形顺序打印二叉树》]

60. 从上到下按层打印二叉树，同一层结点从左至右输出。每一层输出一行。

```
    题目：

    从上到下按层打印二叉树，同一层的结点按从左到右的顺序打印，每一层打印一行。


    思路：

    用一个队列来保存将要打印的结点。为了把二叉树的每一行单独打印到一行里，我们需要两个变量：一个变量表示在当前的层中还
    没有打印的结点数，另一个变量表示下一次结点的数目。
```

61. 序列化二叉树（待深入理解）

```
    题目：

    请实现两个函数，分别用来序列化和反序列化二叉树。


    思路：

    数组模拟
```

62. 二叉搜索树的第 K 个节点

```
    题目：

    给定一颗二叉搜索树，请找出其中的第 k 小的结点。


    思路：

    对一颗树首先进行中序遍历，在遍历的同时记录已经遍历的节点数，当遍历到第 k 个节点时，这个节点即为第 k 大的节点。
```

63. 数据流中的中位数（待深入理解）

```
    题目：

    如何得到一个数据流中的中位数？如果从数据流中读出奇数个数值，那么中位数就是所有值排序之后位于中间的数值。如果数据
    流中读出偶数个数值，那么中位数就是所有数值排序之后中间两个数的平均值。

```

64. 滑动窗口中的最大值（待深入理解）

```
    题目：

    给定一个数组和滑动窗口的大小，找出所有滑动窗口里数值的最大值。例如，如果输入数组{2,3,4,2,6,2,5,1}及滑动窗口的
    大小3，那么一共存在6个滑动窗口，他们的最大值分别为{4,4,6,6,6,5}； 针对数组{2,3,4,2,6,2,5,1}的滑动窗口有以下
    6个： {[2,3,4],2,6,2,5,1}， {2,[3,4,2],6,2,5,1}， {2,3,[4,2,6],2,5,1}， {2,3,4,[2,6,2],5,1}， {2
    ,3,4,2,[6,2,5],1}， {2,3,4,2,6,[2,5,1]}。


    思路：

    使用队列的方式模拟
```

65. 矩阵中的路径（待深入理解）

```
    题目：

    请设计一个函数，用来判断在一个矩阵中是否存在一条包含某字符串所有字符的路径。路径可以从矩阵中的任意一个格子开始，每
    一步可以在矩阵中向左，向右，向上，向下移动一个格子。如果一条路径经过了矩阵中的某一个格子，则该路径不能再进入该格子
    。例如 a b c e s f c s a d e e 矩阵中包含一条字符串"bcced"的路径，但是矩阵中不包含"abcb"路径，因为字符串的
    第一个字符 b 占据了矩阵中的第一行第二个格子之后，路径不能再次进入该格子。
```

66. 机器人的运动范围（待深入理解）

```
    题目：

    地上有一个m行和n列的方格。一个机器人从坐标0,0的格子开始移动，每一次只能向左，右，上，下四个方向移动一格，但是不能
    进入行坐标和列坐标的数位之和大于k的格子。 例如，当k为18时，机器人能够进入方格（35,37），因为3+5+3+7 = 18。但是
    ，它不能进入方格（35,38），因为3+5+3+8 = 19。请问该机器人能够达到多少个格子?
```

剑指 offer 相关资料可以参考：
《剑指 offer 题目练习及思路分析》
《JS 版剑指 offer》
《剑指 Offer 学习心得》

相关算法题

1. 明星问题

```
    题目：

    有 n 个人，其中一个明星和 n-1 个群众，群众都认识明星，明星不认识任何群众，群众和群众之间的认识关系不知道，现有一个
    函数 foo(A, B)，若 A 认识 B 返回 true，若 A 不认识 B 返回 false，试设计一种算法找出明星，并给出时间复杂度。


    思路：

    （1）第一种方法我们可以直接使用双层循环遍历的方式，每一个人都和其他人进行判断，如果一个人谁都不认识，那么他就是明星。
        这一种方法的时间复杂度为 O(n^2)。

    （2）上一种方法没有充分利用题目所给的条件，其实我们每一次比较，都可以排除一个人的可能。比如如果 A 认识 B，那么说明
```

```
        A 就不会是明星，因此 A 就可以从数组中移除。如果 A 不认识 B，那么说明 B 不可能是明星，因此 B 就可以从数组中移
        除。因此每一次判断都能够减少一个可能性，我们只需要从数组从前往后进行遍历，每次移除一个不可能的人，直到数组中只剩
        一人为止，那么这个人就是明星。这一种方法的时间复杂度为 O(n)。
```

详细资料可以参考：
《一个明星和 n-1 个群众》

2. 正负数组求和

```
    题目：

    有两个数组，一个数组里存放的是正整数，另一个数组里存放的是负整数，都是无序的，现在从两个数组里各拿一个，使得它们的和
    最接近零。


    思路：

    （1）首先我们可以对两个数组分别进行排序，正数数组按从小到大排序，负数数组按从大到小排序。排序完成后我们使用两个指针分
        别指向两个数组的首部，判断两个指针的和。如果和大于0，则负数指针往后移动一个位置，如果和小于0，则正数指针往后移动
        一个位置，每一次记录和的值，和当前保存下来的最小值进行比较。
```