什么是数据结构?

作者：Gao-Eason
原文链接：https://ld246.com/article/1649670695193
来源网站：链滴

<!-- /TOC -->

什么是数据结构?

简单地说，数据结构是以某种特定的布局方式存储数据的容器。这种“布局方式”决定了数据结构对某些操作是高效的，而对于其他操作则是低效的。首先我们需要理解各种数据结构，才能在处理实际题时选取最合适的数据结构。

为什么我们需要数据结构?

数据是计算机科学当中最关键的实体，而数据结构则可以将数据以某种组织形式存储，因此，数据结的价值不言而喻。

无论你以何种方式解决何种问题，你都需要处理数据——无论是涉及员工薪水、股票价格、购物清单还是只是简单的电话簿问题。

数据需要根据不同的场景，按照特定的格式进行存储。有很多数据结构能够满足以不同格式存储数据需求。

常见的数据结构

首先列出一些最常见的数据结构，我们将逐一说明:

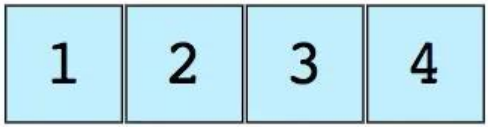
- **数组**
- **栈**
- **队列**
- **链表**
- **树**

原文链接：什么是数据结构?

- 图
- 字典树（这是一种高效的树形结构，但值得单独说明）
- 散列表（哈希表）

1. 数组

数组是最简单、也是使用最广泛的数据结构。栈、队列等其他数据结构均由数组演变而来。下图是一包含元素（1，2，3和4）的简单数组，数组长度为4。



每个数据元素都关联一个正数值，我们称之为索引，它表明数组中每个元素所在的位置。大部分语言初始索引定义为零。关注Java技术栈微信公众号，回复"面试"获取更多博主精心整理的面试题。

以下是数组的两种类型：

- 一维数组（如上所示）
- 多维数组（数组的数组）

数组的基本操作

- Insert——在指定索引位置插入一个元素
- Get——返回指定索引位置的元素
- Delete——删除指定索引位置的元素
- Size——得到数组所有元素的数量

面试中关于数组的常见问题

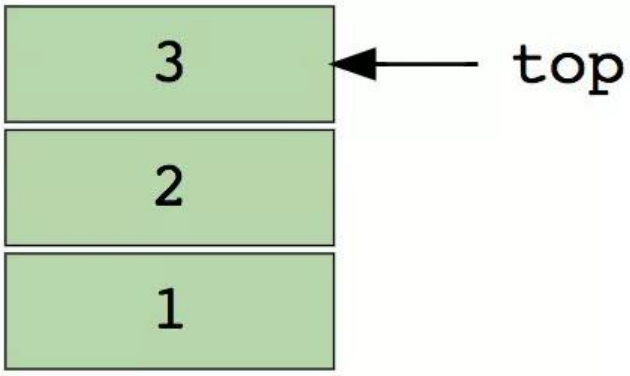
- 寻找数组中第二小的元素
- 找到数组中第一个不重复出现的整数
- 合并两个有序数组
- 重新排列数组中的正值和负值

2. 栈

著名的撤销操作几乎遍布任意一个应用。但你有没有思考过它是如何工作的呢？这个问题的解决思路按照将最后的状态排列在先的顺序，在内存中存储历史工作状态（当然，它会受限于一定的数量）。没办法用数组实现。但有了栈，这就变得非常方便了。

可以把栈想象成一列垂直堆放的书。为了拿到中间的书，你需要移除放置在这上面的所有书。这就是LFO（后进先出）的工作原理。

下图是包含三个数据元素（1，2和3）的栈，其中顶部的3将被最先移除：



栈的基本操作

- **Push——在顶部插入一个元素**
- **Pop——返回并移除栈顶元素**
- **isEmpty——如果栈为空，则返回true**
- **Top——返回顶部元素，但并不移除它**

面试中关于栈的常见问题

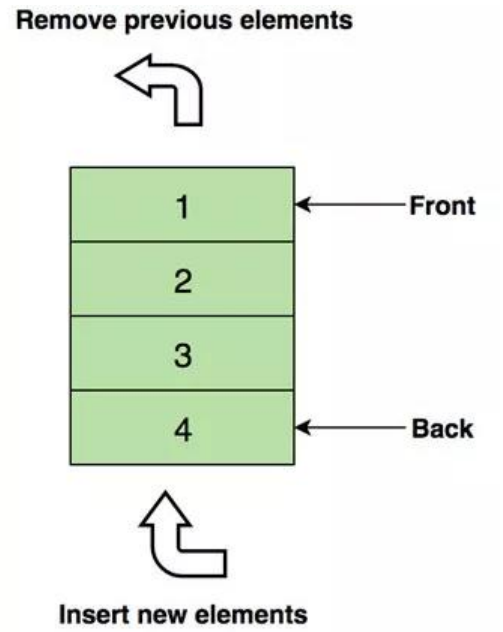
- **使用栈计算后缀表达式**
- **对栈的元素进行排序**
- **判断表达式是否括号平衡**

3. 队列

与栈相似，队列是另一种顺序存储元素的线性数据结构。栈与队列的最大差别在于栈是LIFO（后进先），而队列是FIFO，即先进先出。

一个完美的队列现实例子：售票亭排队队伍。如果有新人加入，他需要到队尾去排队，而非队首——在前面的人会先拿到票，然后离开队伍。

下图是包含四个元素（1，2，3和4）的队列，其中在顶部的1将被最先移除：



原文链接：什么是数据结构?

移除先入队的元素、插入新元素

队列的基本操作

- Enqueue()——在队列尾部插入元素
- Dequeue()——移除队列头部的元素
- isEmpty()——如果队列为空，则返回true
- Top()——返回队列的第一个元素

面试中关于队列的常见问题

- 使用队列表示栈
- 对队列的前k个元素倒序
- 使用队列生成从1到n的二进制数

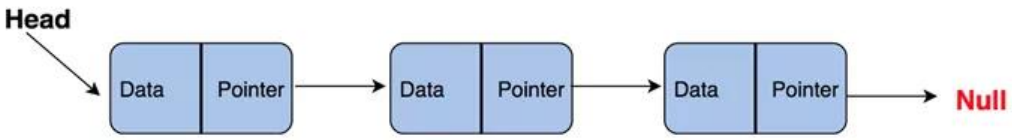
4. 链表

链表是另一个重要的线性数据结构，乍一看可能有点像数组，但在内存分配、内部结构以及数据插入删除的基本操作方面均有所不同。关注Java技术栈微信公众号，回复"面试"获取更多博主精心整理的试题。

链表就像一个节点链，其中每个节点包含着数据和指向后续节点的指针。 链表还包含一个头指针，它向链表的第一个元素，但当列表为空时，它指向null或无具体内容。

链表一般用于实现文件系统、哈希表和邻接表。

这是链表内部结构的展示：



链表包括以下类型：

- 单链表（单向）
- 双向链表（双向）

链表的基本操作：

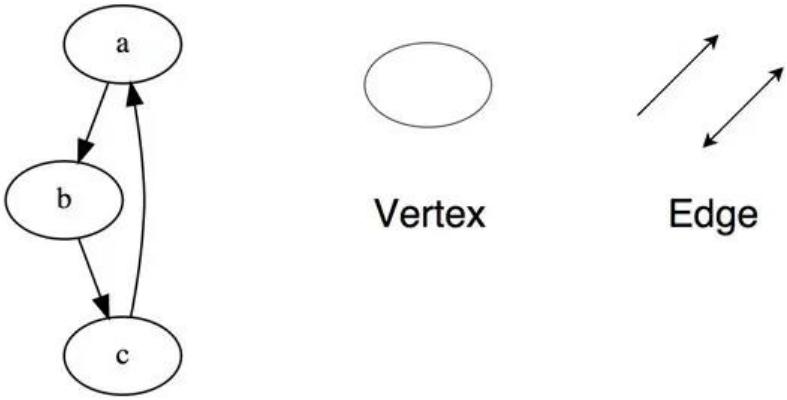
- InsertAtEnd - 在链表的末尾插入指定元素
- InsertAtHead - 在链接列表的开头/头部插入指定元素
- Delete - 从链接列表中删除指定元素
- DeleteAtHead - 删除链接列表的第一个元素
- Search - 从链表中返回指定元素
- isEmpty - 如果链表为空，则返回true

面试中关于链表的常见问题

- **反转链表**
- **检测链表中的循环**
- **返回链表倒数第N个节点**
- **删除链表中的重复项**

5. **图**

图是一组以网络形式相互连接的节点。节点也称为顶点。 一对节点（x，y）称为边（edge），表示点x连接到顶点y。边可以包含权重/成本，显示从顶点x到y所需的成本。



图的类型

- **无向图**
- **有向图**

在程序语言中，图可以用两种形式表示：

- **邻接矩阵**
- **邻接表**

常见图遍历算法

- **广度优先搜索**
- **深度优先搜索**

面试中关于图的常见问题

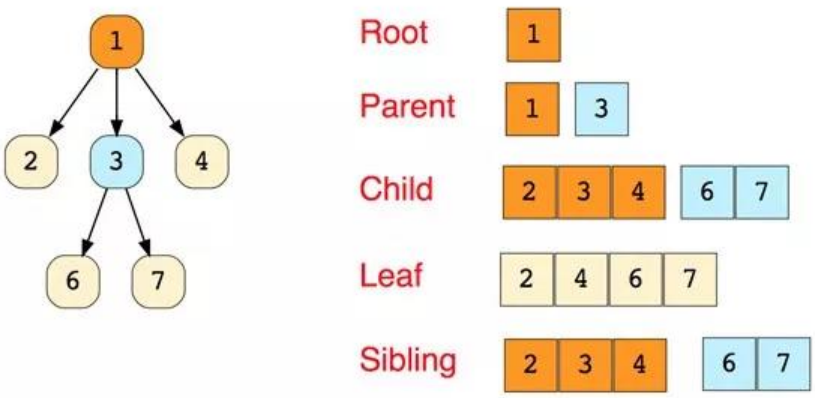
- **实现广度和深度优先搜索**
- **检查图是否为树**
- **计算图的边数**
- **找到两个顶点之间的最短路径**

6. 树

树形结构是一种层级式的数据结构，由顶点（节点）和连接它们的边组成。 树类似于图，但区分树和的重要特征是树中不存在环路。

树形结构被广泛应用于人工智能和复杂算法，它可以提供解决问题的有效存储机制。

这是一个简单树的示意图，以及树数据结构中使用的基本术语：



Root - 根节点

Parent - 父节点

Child - 子节点

Leaf - 叶子节点

Sibling - 兄弟节点

以下是树形结构的主要类型：

- **N元树**
- **平衡树**
- **二叉树**
- **二叉搜索树**
- **AVL树**
- **红黑树**
- **2-3树**

其中，二叉树和二叉搜索树是最常用的树。

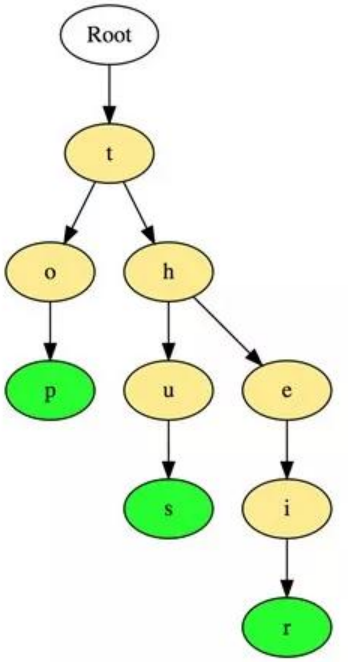
面试中关于树结构的常见问题：

- **求二叉树的高度**
- **在二叉搜索树中查找第k个最大值**
- **查找与根节点距离k的节点**
- **在二叉树中查找给定节点的祖先节点**

7. 字典树（Trie）

字典树，也称为“前缀树”，是一种特殊的树状数据结构，对于解决字符串相关问题非常有效。它能提供快速检索，主要用于搜索字典中的单词，在搜索引擎中自动提供建议，甚至被用于IP的路由。

以下是在字典树中存储三个单词“top”，“so”和“their”的例子：



这些单词以顶部到底部的方式存储，其中绿色节点“p”，“s”和“r”分别表示“top”，“thus和“theirs”的底部。

面试中关于字典树的常见问题

- 计算字典树中的总单词数
- 打印存储在字典树中的所有单词
- 使用字典树对数组的元素进行排序
- 使用字典树从字典中形成单词
- 构建T9字典（字典树+ DFS ）

8. 哈希表

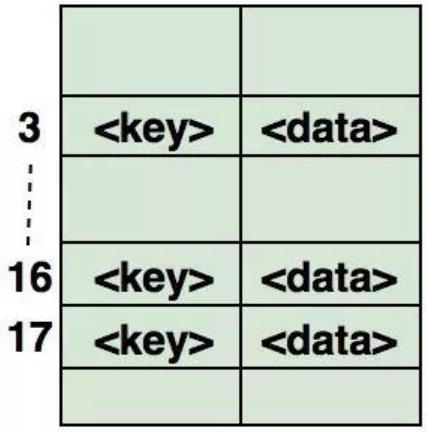
哈希法（Hashing）是一个用于唯一标识对象并将每个对象存储在一些预先计算的唯一索引（称为“（key）”）中的过程。因此，对象以键值对的形式存储，这些键值对的集合被称为“字典”。可以用键搜索每个对象。基于哈希法有很多不同的数据结构，但最常用的数据结构是哈希表。

哈希表通常使用数组实现。

散列数据结构的性能取决于以下三个因素：

- 哈希函数
- 哈希表的大小
- 碰撞处理方法

下图为如何在数组中映射哈希键值对的说明。该数组的索引是通过哈希函数计算的。



面试中关于哈希结构的常见问题：

- **在数组中查找对称键值对**
- **追踪遍历的完整路径**
- **查找数组是否是另一个数组的子集**
- **检查给定的数组是否不相交**

冒泡排序

定义一个布尔变量 hasChange，用来标记每轮是否进行了交换。在每轮遍历开始时，将 hasChange 设置为 false。

若当轮没有发生交换，说明此时数组已经按照升序排列，hashChange** 依然是为 false。此时外层环直接退出，排序结束。**

代码示例

```
import java.util.Arrays;

public class BubbleSort {

    private static void bubbleSort(int[] nums) {
        boolean hasChange = true;
        for (int i = 0, n = nums.length; i < n - 1 && hasChange; ++i) {
            hasChange = false;
            for (int j = 0; j < n - i - 1; ++j) {
                if (nums[j] > nums[j + 1]) {
                    swap(nums, j, j + 1);
                    hasChange = true;
                }
            }
        }
    }

    private static void swap(int[] nums, int i, int j) {
        int t = nums[i];
        nums[i] = nums[j];
        nums[j] = t;
    }
```

```
    public static void main(String[] args) {
        int[] nums = {1, 2, 7, 9, 5, 8};
        bubbleSort(nums);
        System.out.println(Arrays.toString(nums));
    }
}
```

算法分析

空间复杂度 O(1)、时间复杂度 O(n²)。

分情况讨论：

1. **给定的数组按照顺序已经排好：只需要进行 n-1 次比较，两两交换次数为 0，时间复杂度为 O(n) 这是最好的情况。**
2. **给定的数组按照逆序排列：需要进行 n*(n-1)/2 次比较，时间复杂度为 O(n²)，这是最坏的情况。**
3. **给定的数组杂乱无章。在这种情况下，平均时间复杂度 O(n²)。**

因此，时间复杂度是 O(n²)，这是一种稳定的排序算法。

稳定是指，两个相等的数，在排序过后，相对位置保持不变。

插入排序

先来看一个问题。一个有序的数组，我们往里面添加一个新的数据后，如何继续保持数据有序呢？很单，我们只要遍历数组，找到数据应该插入的位置将其插入即可。

这是一个动态排序的过程，即动态地往有序集合中添加数据，我们可以通过这种方法保持集合中的数一直有序。而对于一组静态数据，我们也可以借鉴上面讲的插入方法，来进行排序，于是就有了插入序算法。

那么插入排序具体是如何借助上面的思想来实现排序的呢？

首先，我们将数组中的数据分为两个区间，已排序区间和
排序区间。初始已排序区间只有一个元素，就是数组的第一个元素。插入算法的核心思想是取未排区间中的元素，在已排序区间中找到合适的插入位置将其插入，并保证已排序区间数据一直有序。重这个过程，直到未排序区间中元素为空，算法结束。**

与冒泡排序对比：

- **在冒泡排序中，经过每一轮的排序处理后，数组后端的数是排好序的。**
- **在插入排序中，经过每一轮的排序处理后，数组前端的数是排好序的。**

代码示例

```
import java.util.Arrays;

public class InsertionSort {

    private static void insertionSort(int[] nums) {
        for (int i = 1, j, n = nums.length; i < n; ++i) {
            int num = nums[i];
```

```
            for (j = i - 1; j >=0 && nums[j] > num; --j) {
                nums[j + 1] = nums[j];
            }
            nums[j + 1] = num;
        }
    }

    public static void main(String[] args) {
        int[] nums = {1, 2, 7, 9, 5, 8};
        insertionSort(nums);
        System.out.println(Arrays.toString(nums));
    }
}
```

算法分析

空间复杂度 O(1)，时间复杂度 O(n²)。

分情况讨论：

1. **给定的数组按照顺序排好序：只需要进行 n-1 次比较，两两交换次数为 0，时间复杂度为 O(n)，是最好的情况。**
2. **给定的数组按照逆序排列：需要进行 n*(n-1)/2 次比较，时间复杂度为 O(n²)，这是最坏的情况。**
3. **给定的数组杂乱无章：在这种情况下，平均时间复杂度是 O(n²)。**

因此，时间复杂度是 O(n²)，这也是一种稳定的排序算法。

选择排序

选择排序算法的实现思路有点类似插入排序，也分已排序区间和未排序区间。但是选择排序每次会从排序区间中找到最小的元素，将其放到已排序区间的末尾。

代码示例

```
import java.util.Arrays;

public class SelectionSort {

    private static void selectionSort(int[] nums) {
        for (int i = 0, n = nums.length; i < n - 1; ++i) {
            int minIndex = i;
            for (int j = i; j < n; ++j) {
                if (nums[j] < nums[minIndex]) {
                    minIndex = j;
                }
            }
            swap(nums, minIndex, i);
        }
    }

    private static void swap(int[] nums, int i, int j) {
        int t = nums[i];
        nums[i] = nums[j];
```

```
        nums[j] = t;
    }

    public static void main(String[] args) {
        int[] nums = {1, 2, 7, 9, 5, 8};
        selectionSort(nums);
        System.out.println(Arrays.toString(nums));
    }
}
```

算法分析

空间复杂度 O(1)，时间复杂度 O(n^2)。

那选择排序是稳定的排序算法吗?

答案是否定的，选择排序是一种不稳定的排序算法。选择排序每次都要找剩余未排序元素中的最小值，并和前面的元素交换位置，这样破坏了稳定性。

比如 5，8，5，2，9 这样一组数据，使用选择排序算法来排序的话，第一次找到最小元素 2，与第一个 5 交换位置，那第一个 5 和中间的 5 顺序就变了，所以就不稳定了。正是因此，相对于冒泡排序和插入排序，选择排序就稍微逊色了。

归并排序

归并排序的核心思想是分治，把一个复杂问题拆分成若干个子问题来求解。

归并排序的算法思想是：把数组从中间划分为两个子数组，一直递归地把子数组划分成更小的数组，到子数组里面只有一个元素的时候开始排序。排序的方法就是按照大小顺序合并两个元素。接着依次照递归的顺序返回，不断合并排好序的数组，直到把整个数组排好序。

代码示例

```
import java.util.Arrays;

public class MergeSort {

    private static void merge(int[] nums, int low, int mid, int high, int[] temp) {
        int i = low, j = mid + 1, k = low;
        while (k <= high) {
            if (i > mid) {
                temp[k++] = nums[j++];
            } else if (j > high) {
                temp[k++] = nums[i++];
            } else if (nums[i] <= nums[j]) {
                temp[k++] = nums[i++];
            } else {
                temp[k++] = nums[j++];
            }
        }

        System.arraycopy(tmp, low, nums, low, high - low + 1);
    }
```

```
    private static void mergeSort(int[] nums, int low, int high, int[] temp) {
        if (low >= high) {
            return;
        }
        int mid = (low + high) >>> 1;
        mergeSort(nums, low, mid, temp);
        mergeSort(nums, mid + 1, high, temp);
        merge(nums, low, mid, high, temp);
    }

    private static void mergeSort(int[] nums) {
        int n = nums.length;
        int[] temp = new int[n];
        mergeSort(nums, 0, n - 1, temp);
    }

    public static void main(String[] args) {
        int[] nums = {1, 2, 7, 4, 5, 3};
        mergeSort(nums);
        System.out.println(Arrays.toString(nums));
    }
}
```

算法分析

空间复杂度 O(n)，时间复杂度 O(nlogn)。

对于规模为 n 的问题，一共要进行 log(n) 次的切分，每一层的合并复杂度都是 O(n)，所以整体时复杂度为 O(nlogn)。

由于合并 n 个元素需要分配一个大小为 n 的额外数组，所以空间复杂度为 O(n)。

这是一种稳定的排序算法。

快速排序

快速排序也采用了分治的思想：把原始的数组筛选成较小和较大的两个子数组，然后递归地排序两个数组。

代码示例

```
import java.util.Arrays;

public class QuickSort {

    private static void quickSort(int[] nums) {
        quickSort(nums, 0, nums.length - 1);
    }

    private static void quickSort(int[] nums, int low, int high) {
        if (low >= high) {
            return;
        }
        int[] p = partition(nums, low, high);
```

```
        quickSort(nums, low, p[0] - 1);
        quickSort(nums, p[0] + 1, high);
    }

    private static int[] partition(int[] nums, int low, int high) {
        int less = low - 1, more = high;
        while (low < more) {
            if (nums[low] < nums[high]) {
                swap(nums, ++less, low++);
            } else if (nums[low] > nums[high]) {
                swap(nums, --more, low);
            } else {
                ++low;
            }
        }
        swap(nums, more, high);
        return new int[] {less + 1, more};
    }

    private static void swap(int[] nums, int i, int j) {
        int t = nums[i];
        nums[i] = nums[j];
        nums[j] = t;
    }

    public static void main(String[] args) {
        int[] nums = {1, 2, 7, 4, 5, 3};
        quickSort(nums);
        System.out.println(Arrays.toString(nums));
    }
}
```

算法分析

空间复杂度 O(logn)，时间复杂度 O(nlogn)。

对于规模为 n 的问题，一共要进行 log(n) 次的切分，和基准值进行 n-1 次比较，n-1 次比较的时间杂度是 O(n)，所以快速排序的时间复杂度为 O(nlogn)。

但是，如果每次在选择基准值的时候，都不幸地选择了子数组里的最大或最小值。即每次把把数组分了两个更小长度的数组，其中一个长度为 1，另一个的长度是子数组的长度减 1。这样的算法复杂度成 $O(n^2)$。

和归并排序不同，快速排序在每次递归的过程中，只需要开辟 O(1) 的存储空间来完成操作来实现对组的修改；而递归次数为 logn，所以它的整体空间复杂度完全取决于压堆栈的次数。

如何优化快速排序?

前面讲到，最坏情况下快速排序的时间复杂度是 $O(n^2)$，实际上，这种 $O(n^2)$ 时间复杂度出现的主要因还是因为我们基准值选得不够合理。最理想的基准点是：被基准点分开的两个子数组中，数据的数差不多。

如果很粗暴地直接选择第一个或者最后一个数据作为基准值，不考虑数据的特点，肯定会出现之前讲那样，在某些情况下，排序的最坏情况时间复杂度是 $O(n^2)$。

有两个比较常用的分区算法。

1. **三数取中法**

我们从区间的首、尾、中间，分别取出一个数，然后对比大小，取这 3 个数的中间值作为分区点。这每间隔某个固定的长度，取数据出来比较，将中间值作为分区点的分区算法，肯定要比单纯取某一个据更好。但是，如果要排序的数组比较大，那“三数取中”可能就不够了，可能要“五数取中”或者十数取中”。

2. **随机法**

随机法就是每次从要排序的区间中，随机选择一个元素作为分区点。这种方法并不能保证每次分区点选的比较好，但是从概率的角度来看，也不大可能会出现每次分区点都选的很差的情况，所以平均情下，这样选的分区点是比较好的。时间复杂度退化为最糟糕的 O(n^2) 的情况，出现的可能性不大。

二分查找

二分查找是一种非常高效的查找算法，高效到什么程度呢？我们来分析一下它的时间复杂度。

假设数据大小是 n，每次查找后数据都会缩小为原来的一半，也就是会除以 2。最坏情况下，直到查区间被缩小为空，才停止。

被查找区间的大小变化为：

n, n/2, n/4, n/8, ..., n/(2^k)

可以看出来，这是一个等比数列。其中 n/(2^k)=1 时，k 的值就是总共缩小的次数。而每一次缩小作只涉及两个数据的大小比较，所以，经过了 k 次区间缩小操作，时间复杂度就是 O(k)。通过 n/(2 k)=1，我们可以求得 k=log2n，所以时间复杂度就是 O(logn)。

代码示例

注意容易出错的 3 个地方。

1. **循环退出条件是 low <= high，而不是 low < high；**
2. **mid 的取值，可以是 mid = (low + high) / 2，但是如果 low 和 high 比较大的话，** low + high* 可能会溢出，所以这里写为 **mid = (low + high) >>> 1**；**
3. **low 和 high 的更新分别为 low = mid + 1、** high = mid - 1**。**

非递归实现：

```
public class BinarySearch {

  private static int search(int[] nums, int low, int high, int val) {
    while (low <= high) {
      int mid = (low + high) >>> 1;
      if (nums[mid] == val) {
        return mid;
      } else if (nums[mid] < val) {
        low = mid + 1;
      } else {
        high = mid - 1;
      }
```

```
        }
        return -1;
    }

    /**
     * 二分查找(非递归)
     *
     * @param nums 有序数组
     * @param val 要查找的值
     * @return 要查找的值在数组中的索引位置
     */
    private static int search(int[] nums, int val) {
        return search(nums, 0, nums.length - 1, val);
    }

    public static void main(String[] args) {
        int[] nums = {1, 2, 5, 7, 8, 9};

        // 非递归查找
        int r1 = search(nums, 7);
        System.out.println(r1);
    }
}
```

递归实现：

```
public class BinarySearch {

    private static int searchRecursive(int[] nums, int low, int high, int val) {
        while (low <= high) {
            int mid = (low + high) >>> 1;
            if (nums[mid] == val) {
                return mid;
            } else if (nums[mid] < val) {
                return searchRecursive(nums, mid + 1, high, val);
            } else {
                return searchRecursive(nums, low, mid - 1, val);
            }
        }
        return -1;
    }

    /**
     * 二分查找(递归)
     *
     * @param nums 有序数组
     * @param val 要查找的值
     * @return 要查找的值在数组中的索引位置
     */
    private static int searchRecursive(int[] nums, int val) {
        return searchRecursive(nums, 0, nums.length - 1, val);
    }

    public static void main(String[] args) {
```

```
        int[] nums = {1, 2, 5, 7, 8, 9};

        // 递归查找
        int r2 = searchRecursive(nums, 7);
        System.out.println(r2);
    }
}
```

二分查找 II

前面讲的二分查找算法，是最为简单的一种，在不存在重复元素的有序数组中，查找值等于给定值的素。

接下来，我们来看看二分查找算法四种常见的变形问题，分别是：

1. **查找第一个值等于给定值的元素**
2. **查找最后一个值等于给定值的元素**
3. **查找第一个大于等于给定值的元素**
4. **查找最后一个小于等于给定值的元素**

1、查找第一个值等于给定值的元素

```
public static int search(int[] nums, int val) {
    int n = nums.length;
    int low = 0, high = n - 1;
    while (low <= high) {
        int mid = (low + high) >>> 1;
        if (nums[mid] < val) {
            low = mid + 1;
        } else if (nums[mid] > val) {
            high = mid - 1;
        } else {
            // 如果nums[mid]是第一个元素，或者nums[mid-1]不等于val
            // 说明nums[mid]就是第一个值为给定值的元素
            if (mid == 0 || nums[mid - 1] != val) {
                return mid;
            }
            high = mid - 1;
        }
    }
    return -1;
}
```

2、查找最后一个值等于给定值的元素

```
public static int search(int[] nums, int val) {
    int n = nums.length;
    int low = 0, high = n - 1;
    while (low <= high) {
        int mid = (low + high) >>> 1;
        if (nums[mid] < val) {
            low = mid + 1;
```

```
        } else if (nums[mid] > val) {
            high = mid - 1;
        } else {
            // 如果nums[mid]是最后一个元素，或者nums[mid+1]不等于val
            // 说明nums[mid]就是最后一个值为给定值的元素
            if (mid == n - 1 || nums[mid + 1] != val) {
                return mid;
            }
            low = mid + 1;
        }
    }
    return -1;
}
```

3、查找第一个大于等于给定值的元素

```
public static int search(int[] nums, int val) {
    int low = 0, high = nums.length - 1;
    while (low <= high) {
        int mid = (low + high) >>> 1;
        if (nums[mid] < val) {
            low = mid + 1;
        } else {
            // 如果nums[mid]是第一个元素，或者nums[mid-1]小于val
            // 说明nums[mid]就是第一个大于等于给定值的元素
            if (mid == 0 || nums[mid - 1] < val) {
                return mid;
            }
            high = mid - 1;
        }
    }
    return -1;
}
```

4、查找最后一个小于等于给定值的元素

```
public static int search(int[] nums, int val) {
    int n = nums.length;
    int low = 0, high = n - 1;
    while (low <= high) {
        int mid = (low + high) >>> 1;
        if (nums[mid] > val) {
            high = mid - 1;
        } else {
            // 如果nums[mid]是最后一个元素，或者nums[mid+1]大于val
            // 说明nums[mid]就是最后一个小于等于给定值的元素
            if (mid == n - 1 || nums[mid + 1] > val) {
                return mid;
            }
            low = mid + 1;
        }
    }
    return -1;
}
```

删除排序数组中的重复项

题目描述

给定一个排序数组，你需要在原地删除重复出现的元素，使得每个元素出现一次，返回移除后数组的新长度。

不要使用额外的数组空间，你必须在 原地修改输入数组 并在使用 O(1) 额外空间的条件下完成。

示例 1:

给定数组 nums = [1,1,2],

函数应该返回新的长度 2, 并且原数组 nums 的前两个元素被修改为 1, 2。

你不需要考虑数组中超出新长度后面的元素。

示例 2:

给定 nums = [0,0,1,1,1,2,2,3,3,4],

函数应该返回新的长度 5, 并且原数组 nums 的前五个元素被修改为 0, 1, 2, 3, 4。

你不需要考虑数组中超出新长度后面的元素。

说明:

为什么返回数值是整数，但输出的答案是数组呢?

请注意，输入数组是以**「引用」****方式传递的，这意味着在函数里修改输入数组对于调用者是见的。**

```
class Solution {
    public int removeDuplicates(int[] nums) {
        int cnt = 0, n = nums.length;
        for (int i = 1; i < n; ++i) {
            if (nums[i] == nums[i - 1]) ++cnt;
            else nums[i - cnt] = nums[i];
        }
        return n - cnt;
    }
}
```

删除排序数组中的重复项 II

给定一个排序数组，你需要在原地删除重复出现的元素，使得每个元素多出现两次，返回移除后数组的新长度。

不要使用额外的数组空间，你必须在原地修改输入数组并在使用

(1) 额外空间的条件下完成。

示例 1:

给定 nums = [1,1,1,2,2,3],

函数应返回新长度 length = 5, 并且原数组的前五个元素被修改为 1, 1, 2, 2, 3 。

你不需要考虑数组中超出新长度后面的元素。

示例 2:

给定 nums = [0,0,1,1,1,1,2,3,3],

函数应返回新长度 length = 7, 并且原数组的前五个元素被修改为 0, 0, 1, 1, 2, 3, 3 。

你不需要考虑数组中超出新长度后面的元素。

说明:

为什么返回数值是整数，但输出的答案是数组呢?

请注意，输入数组是以****“引用”******方式传递的，这意味着在函数里修改输入数组对于调用者是见的。**

你可以想象内部操作如下:

```
// nums 是以“引用”方式传递的。也就是说，不对实参做任何拷贝
int len = removeDuplicates(nums);

// 在函数里修改输入数组对于调用者是可见的。
// 根据你的函数返回的长度, 它会打印出数组中该长度范围内的所有元素。
for (int i = 0; i < len; i++) {
    print(nums[i]);
}
```

解法

从数组下标 1 开始遍历数组。

用计数器 cnt 记录当前数字重复出现的次数，cnt** 的最小计数为 0；用 cur 记录新数组下个待覆盖元素位置。**

遍历时，若当前元素 nums[i] 与上个元素 nums[i-1] 相同，则计数器 +1，否则计数器重置为 0。果计数器小于 2，说明当前元素 nums[i] 可以添加到新数组中，即：nums[cur] = nums[i]**，同时 cu++。**

遍历结果，返回 cur 值即可。

```
class Solution {
    public int removeDuplicates(int[] nums) {
        int cnt = 0, cur = 1;
        for (int i = 1; i < nums.length; ++i) {
            if (nums[i] == nums[i - 1]) ++cnt;
            else cnt = 0;
```

```
            if (cnt < 2) nums[cur++] = nums[i];
        }
        return cur;
    }
}
```

移除元素

题目描述

给你一个数组 nums 和一个值 val
你需要 原地 移除所有数值等于 val 的元素，并返回移除后数组的新长度。

不要使用额外的数组空间，你必须仅使用 O(1) 额外空间并 **原地 **修改输入数组。

元素的顺序可以改变。你不需要考虑数组中超出新长度后面的元素。

示例 1:

给定 nums = [3,2,2,3], val = 3,

函数应该返回新的长度 2, 并且 nums 中的前两个元素均为 2。

你不需要考虑数组中超出新长度后面的元素。

示例 2:

给定 nums = [0,1,2,2,3,0,4,2], val = 2,

函数应该返回新的长度 5, 并且 nums 中的前五个元素为 0, 1, 3, 0, 4。

注意这五个元素可为任意顺序。

你不需要考虑数组中超出新长度后面的元素。

说明:

为什么返回数值是整数，但输出的答案是数组呢?

请注意，输入数组是以**「引用」****方式传递的，这意味着在函数里修改输入数组对于调用者是见的。**

你可以想象内部操作如下:

```
// nums 是以“引用”方式传递的。也就是说，不对实参作任何拷贝
int len = removeElement(nums, val);

// 在函数里修改输入数组对于调用者是可见的。
// 根据你的函数返回的长度, 它会打印出数组中 该长度范围内 的所有元素。
for (int i = 0; i < len; i++) {
```

```
    print(nums[i]);
}
```

解法

```
class Solution {
    public int removeElement(int[] nums, int val) {
        int cnt = 0, n = nums.length;
        for (int i = 0; i < n; ++i) {
            if (nums[i] == val) {
                ++cnt;
            } else {
                nums[i - cnt] = nums[i];
            }
        }
        return n - cnt;
    }
}
```

移动零

题目描述

给定一个数组 nums，编写一个函数将所有 0 移动到数组的末尾，同时保持非零元素的相对顺序。

示例:

```
[0,1,0,3,12]
```

说明:**

1. **必须在原数组上操作，不能拷贝额外的数组。**
2. **尽量减少操作次数。**

解法

```
class Solution {
    public void moveZeroes(int[] nums) {
        int n;
        if (nums == null || (n = nums.length) < 1) {
            return;
        }
        int zeroCount = 0;
        for (int i = 0; i < n; ++i) {
            if (nums[i] == 0) {
                ++zeroCount;
            } else {
                nums[i - zeroCount] = nums[i];
            }
        }
        while (zeroCount > 0) {
            nums[n - zeroCount--] = 0;
        }
    }
```

```
}
```

数组中重复的数字

题目描述

找出数组中重复的数字。

在一个长度为 n 的数组 nums 里的所有数字都在 0 ~ n-1 的范围内。数组中某些数字是重复的，但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。

示例 1：

```
输入：
[2, 3, 1, 0, 2, 5, 3]
输出：2 或 3
```

限制：

```
2 <= n <= 100000
```

解法

0 ~ n-1 范围内的数，分别还原到对应的位置上，如：数字 2 交换到下标为 2 的位置。

若交换过程中发现重复，则直接返回。

```
class Solution {
    public int findRepeatNumber(int[] nums) {
        for (int i = 0, n = nums.length; i < n; ++i) {
            while (nums[i] != i) {
                if (nums[i] == nums[nums[i]]) return nums[i];
                swap(nums, i, nums[i]);
            }
        }
        return -1;
    }

    private void swap(int[] nums, int i, int j) {
        int t = nums[i];
        nums[i] = nums[j];
        nums[j] = t;
    }
}
```

旋转数组

题目描述

给定一个数组，将数组中的元素向右移动 k 个位置，其中 k 是非负数。

示例 1：

[1,2,3,4,5,6,7]

示例 2:

[-1,-100,3,99]

说明:

- **尽可能想出更多的解决方案，至少有三种不同的方法可以解决这个问题。**
- **要求使用空间复杂度为 O(1) 的 原地 算法。**

解法

若 k=3，nums=[1,2,3,4,5,6,7]**。**

先将 nums 整体翻转：[1,2,3,4,5,6,7]** -> **[7,6,5,4,3,2,1]

再翻转 0~k-1 范围内的元素：[7,6,5,4,3,2,1]** -> **[5,6,7,4,3,2,1]

最后翻转 k~n-1 范围内的元素，即可得到最终结果：[5,6,7,4,3,2,1]** -> **[5,6,7,1,2,3,4]

```
class Solution {
    public void rotate(int[] nums, int k) {
        if (nums == null) {
            return;
        }
        int n = nums.length;
        k %= n;
        if (n < 2 || k == 0) {
            return;
        }

        rotate(nums, 0, n - 1);
        rotate(nums, 0, k - 1);
        rotate(nums, k, n - 1);
    }

    private void rotate(int[] nums, int i, int j) {
        while (i < j) {
            int t = nums[i];
            nums[i] = nums[j];
            nums[j] = t;
            ++i;
            --j;
        }
    }
}
```

螺旋矩阵

题目描述

给定一个包含 m x n 个元素的矩阵（m 行, n 列），请按照顺时针螺旋顺序，返回矩阵中的所有元素。

示例 1:

```
输入:
[
 [ 1, 2, 3 ],
 [ 4, 5, 6 ],
 [ 7, 8, 9 ]
]
输出: [1,2,3,6,9,8,7,4,5]
```

示例 2:

```
输入:
[
  [1, 2, 3, 4],
  [5, 6, 7, 8],
  [9,10,11,12]
]
输出: [1,2,3,4,8,12,11,10,9,5,6,7]
```

提示:**

- **m == matrix.length**
- **n == matrix[i].length**
- **1 <= m, n <= 10**
- **-100 <= matrix**i** <= 100**

解法

从外往里一圈一圈遍历并存储矩阵元素即可。

```
class Solution {
    private List<Integer> res;

    public List<Integer> spiralOrder(int[][] matrix) {
        int m = matrix.length, n = matrix[0].length;
        res = new ArrayList<>();
        int i1 = 0, i2 = m - 1;
        int j1 = 0, j2 = n - 1;
        while (i1 <= i2 && j1 <= j2) {
            add(matrix, i1++, j1++, i2--, j2--);
        }
        return res;
    }

    private void add(int[][] matrix, int i1, int j1, int i2, int j2) {
        if (i1 == i2) {
            for (int j = j1; j <= j2; ++j) {
                res.add(matrix[i1][j]);
            }
            return;
        }
        if (j1 == j2) {
```

```
            for (int i = i1; i <= i2; ++i) {
                res.add(matrix[i][j1]);
            }
            return;
        }
        for (int j = j1; j < j2; ++j) {
            res.add(matrix[i1][j]);
        }
        for (int i = i1; i < i2; ++i) {
            res.add(matrix[i][j2]);
        }
        for (int j = j2; j > j1; --j) {
            res.add(matrix[i2][j]);
        }
        for (int i = i2; i > i1; --i) {
            res.add(matrix[i][j1]);
        }
    }
}
```

两数之和

题目描述

给定一个整数数组 nums 和一个目标值 target，请你在该数组中找出和为目标值的那 两个整数，并返回他们的数组下标。

你可以假设每种输入只会对应一个答案。但是，你不能重复利用这个数组中同样的元素。

示例:

```
给定 nums = [2, 7, 11, 15], target = 9

因为 nums[0] + nums[1] = 2 + 7 = 9
所以返回 [0, 1]
```

解法

用哈希表（字典）存放数组值以及对应的下标。

遍历数组，当发现 target - nums[i] 在哈希表中，说明找到了目标值。

```
class Solution {
    public int[] twoSum(int[] nums, int target) {
        Map<Integer, Integer> map = new HashMap<>();
        for (int i = 0, n = nums.length; i < n; ++i) {
            int num = target - nums[i];
            if (map.containsKey(num)) {
                return new int[]{map.get(num), i};
            }
            map.put(nums[i], i);
        }
        return null;
    }
```

}

三数之和

给你一个包含 **n 个整数的数组 **nums**，判断 **nums** 中是否存在三个元素 *****a，b，c ，*****使得 a + b + c = 0 ？ 请你找出所有满足条件且不重复的三元组。**

******注意：******答案中不可以包含重复的三元组。**

示例：

给定数组 nums = [-1, 0, 1, 2, -1, -4]，

满足要求的三元组集合为：
[
  [-1, 0, 1],
  [-1, -1, 2]
]

解法

“排序 + 双指针”实现。

```
class Solution {
    public List<List<Integer>> threeSum(int[] nums) {
        int n;
        if (nums == null || (n = nums.length) < 3) {
            return Collections.emptyList();
        }
        Arrays.sort(nums);
        List<List<Integer>> res = new ArrayList<>();
        for (int i = 0; i < n - 2; ++i) {
            if (i > 0 && nums[i] == nums[i - 1]) {
                continue;
            }
            int p = i + 1, q = n - 1;
            while (p < q) {
                if (p > i + 1 && nums[p] == nums[p - 1]) {
                    ++p;
                    continue;
                }
                if (q < n - 1 && nums[q] == nums[q + 1]) {
                    --q;
                    continue;
                }
                if (nums[p] + nums[q] + nums[i] < 0) {
                    ++p;
                } else if (nums[p] + nums[q] + nums[i] > 0) {
                    --q;
                } else {
                    res.add(Arrays.asList(nums[p], nums[q], nums[i]));
                    ++p;
```

```
                --q;
            }
        }
    }
    return res;
  }
}
```

四数之和

题目描述

给定一个包含 n 个整数的数组 nums 和一个目标值 target，判断 nums 中是否存在四个元素 a，，c 和 d，使得 a + b + c + d 的值与 target 相等？找出所有满足条件且不重复的四元组。

注意：

答案中不可以包含重复的四元组。

示例：

给定数组 nums = [1, 0, -1, 0, -2, 2]，和 target = 0。

满足要求的四元组集合为：

```
[
  [-1,  0, 0, 1],
  [-2, -1, 1, 2],
  [-2,  0, 0, 2]
]
```

解法

“排序 + 双指针”实现。

```
class Solution {
    public List<List<Integer>> fourSum(int[] nums, int target) {
        int n;
        if (nums == null || (n = (nums.length)) < 4) {
            return Collections.emptyList();
        }
        Arrays.sort(nums);
        List<List<Integer>> res = new ArrayList<>();
        for (int i = 0; i < n - 3; ++i) {
            if (i > 0 && nums[i] == nums[i - 1]) {
                continue;
            }
            for (int j = i + 1; j < n - 2; ++j) {
                if (j > i + 1 && nums[j] == nums[j - 1]) {
                    continue;
                }
                int p = j + 1, q = n - 1;
                while (p < q) {
                    if (p > j + 1 && nums[p] == nums[p - 1]) {
```

```
                ++p;
                continue;
            }
            if (q < n - 1 && nums[q] == nums[q + 1]) {
                --q;
                continue;
            }
            int t = nums[i] + nums[j] + nums[p] + nums[q];
            if (t == target) {
                res.add(Arrays.asList(nums[i], nums[j], nums[p], nums[q]));
                ++p;
                --q;
            } else if (t < target) {
                ++p;
            } else {
                --q;
            }
        }
      }
    }
    return res;
  }
}
```

较小的三数之和

题目描述

给定一个长度为 n 的整数数组和一个目标值 target
寻找能够使条件 nums[i] + nums[j] + nums[k] < target 成立的三元组 i, j, k 个数（0 <= i < j < k < n）。

示例：

[-2,0,1,3]

****进阶：****是否能在 O(n2) 的时间复杂度内解决?**

解法

双指针解决。

```
class Solution {
  public int threeSumSmaller(int[] nums, int target) {
    Arrays.sort(nums);
    int n = nums.length;
    int count = 0;
    for (int i = 0; i < n - 2; ++i) {
      count += threeSumSmaller(nums, i + 1, n - 1, target - nums[i]);
    }
    return count;
  }

  private int threeSumSmaller(int[] nums, int start, int end, int target) {
```

```
        int count = 0;
        while (start < end) {
            if (nums[start] + nums[end] < target) {
                count += (end - start);
                ++start;
            } else {
                --end;
            }
        }
        return count;
    }
}
```

最接近的三数之和

题目描述

给定一个包括 n 个整数的数组 nums 和 一个目标值 target。找出 nums 中的三个整数，使得它们的和与 target 最接近。返回这三个数的和。假定每组输入只存在唯一答案。

例如，给定数组 nums = [-1，2，1，-4], 和 target = 1.

与 target 最接近的三个数的和为 2. (-1 + 2 + 1 = 2).

解法

双指针解决。

```
class Solution {
    public int threeSumClosest(int[] nums, int target) {
        Arrays.sort(nums);
        int res = 0;
        int n = nums.length;
        int diff = Integer.MAX_VALUE;
        for (int i = 0; i < n - 2; ++i) {
            int t = twoSumClosest(nums, i + 1, n - 1, target - nums[i]);
            if (Math.abs(nums[i] + t - target) < diff) {
                res = nums[i] + t;
                diff = Math.abs(nums[i] + t - target);
            }
        }
        return res;
    }

    private int twoSumClosest(int[] nums, int start, int end, int target) {
        int res = 0;
        int diff = Integer.MAX_VALUE;
        while (start < end) {
            int val = nums[start] + nums[end];
            if (val == target) {
                return val;
            }
            if (Math.abs(val - target) < diff) {
```

```
                res = val;
                diff = Math.abs(val - target);
            }
            if (val < target) {
                ++start;
            } else {
                --end;
            }
        }
        return res;
    }
}
```

合并两个有序数组

题目描述

给你两个有序整数数组 nums1 和 nums2
请你将 nums2 合并到 nums1 中
，*使 num1 成为一个有序数组。**

说明:

- **初始化 nums1 和 nums2 的元素数量分别为 m**
和 n 。
- **你可以假设 nums1 有足够的空间（空间大小大于或等于 m + n**
来保存 nums2 中的元素。

示例:

```
输入:
nums1 = [1,2,3,0,0,0], m = 3
nums2 = [2,5,6],       n = 3

输出: [1,2,2,3,5,6]
```

解法

双指针解决。

```
class Solution {
    public void merge(int[] nums1, int m, int[] nums2, int n) {
        int i = m - 1, j = n - 1;
        int k = m + n - 1;
        while (j >= 0) {
            if (i >= 0 && nums1[i] >= nums2[j]) {
                nums1[k--] = nums1[i--];
            } else {
                nums1[k--] = nums2[j--];
```

```
            }
        }
    }
}
```

寻找旋转排序数组中的最小值

题目描述

假设按照升序排序的数组在预先未知的某个点上进行了旋转。

(例如，数组 [0,1,2,4,5,6,7] 可能变为 [4,5,6,7,0,1,2])。

请找出其中最小的元素。

你可以假设数组中不存在重复元素。

示例 1:

```
输入: [3,4,5,1,2]
输出: 1
```

示例 2:

```
输入: [4,5,6,7,0,1,2]
输出: 0
```

解法

二分查找。

若 nums[m] > nums[r]，说明最小值在 m 的右边，否则说明最小值在 m 的左边（包括 m）。

```
class Solution {
    public int findMin(int[] nums) {
        int l = 0, r = nums.length - 1;
        while (l < r) {
            int m = (l + r) >>> 1;
            if (nums[m] > nums[r]) {
                l = m + 1;
            } else {
                r = m;
            }
        }
        return nums[l];
    }
}
```

寻找旋转排序数组中的最小值 II

题目描述

假设按照升序排序的数组在预先未知的某个点上进行了旋转。

(例如，数组 [0,1,2,4,5,6,7] 可能变为 [4,5,6,7,0,1,2])。

请找出其中最小的元素。

注意数组中可能存在重复的元素。

示例 1：

输入: [1,3,5]
输出: 1

示例 2：

输入: [2,2,2,0,1]
输出: 0

说明：

● 允许重复会影响算法的时间复杂度吗？会如何影响，为什么？

```
class Solution {
    public int findMin(int[] nums) {
        int l = 0, r = nums.length - 1;
        while (l < r) {
            int m = (l + r) >>> 1;
            if (nums[m] > nums[r]) {
                l = m + 1;
            } else if (nums[m] < nums[r]) {
                r = m;
            } else {
                --r;
            }
        }
        return nums[l];
    }
}
```

除自身以外数组的乘积

题目描述

给你一个长度为 n 的整数数组 nums，其中 n > 1，返回出数组 output ，其中 output[i] 等于 nums 中除 nums[i] 之外其余各元素的乘积。

示例:

[1,2,3,4]

****提示：****题目数据保证数组之中任意元素的全部前缀元素和后缀（甚至是整个数组）的乘积都在 32 位整数范围内。**

说明:** 请******不要使用除法，******且在 O(n) 时间复杂度内完成此题。**

进阶：**

你可以在常数空间复杂度内完成这个题目吗？ （ 出于对空间复杂度分析的目的，输出数组不被视为 *额外空间。）

解法

```
class Solution {
    public int[] productExceptSelf(int[] nums) {
        int n = nums.length;
        int[] output = new int[n];
        for (int i = 0, left = 1; i < n; ++i) {
            output[i] = left;
            left *= nums[i];
        }
        for (int i = n - 1, right = 1; i >= 0; --i) {
            output[i] *= right;
            right *= nums[i];
        }
        return output;
    }
}
```

无重复字符的最长子串

题目描述

给定一个字符串，请你找出其中不含有重复字符的 最长子串 的长度。

示例 1:

"abc"，所以其

示例 2:

"b"

示例 3:

"wke"

解法

- **定义一个哈希表存放字符及其出现的位置；**
- **定义 i, j 分别表示不重复子串的开始位置和结束位置；**
- **j 向后遍历，若遇到与 [i, j] 区间内字符相同的元素，更新 i 的值，此时 [i, j] 区间内不存在重复字，计算 res 的最大值。**

```
class Solution {
    public int lengthOfLongestSubstring(String s) {
        int res = 0;
```

```
        Map<Character, Integer> chars = new HashMap<>();
        for (int i = 0, j = 0; j < s.length(); ++j) {
            char c = s.charAt(j);
            if (chars.containsKey(c)) {
                // chars.get(c)+1 可能比 i 还小，通过 max 函数来锁住左边界
                // e.g. 在"tmmzuxt"这个字符串中，遍历到最后一步时，最后一个字符't'和第一个字符't'
相等的。如果没有 max 函数，i 就会回到第一个't'的索引0处的下一个位置
                i = Math.max(i, chars.get(c) + 1);
            }
            chars.put(c, j);
            res = Math.max(res, j - i + 1);
        }
        return res;
    }
}
```

反转字符串中的元音字母

题目描述

编写一个函数，以字符串作为输入，反转该字符串中的元音字母。

示例 1:

```
输入: "hello"
输出: "holle"
```

示例 2:

```
输入: "leetcode"
输出: "leotcede"
```

说明:**

****元音字母不包含字母"y"。**

解法

将字符串转为字符数组（或列表），定义双指针 p、q，分别指向数组（列表）头部和尾部，当 p、q 指向的字符均为元音字母时，进行交换。

依次遍历，当 p >= q 时，遍历结束。将字符数组（列表）转为字符串返回即可。

```
class Solution {
    public String reverseVowels(String s) {
        if (s == null) {
            return s;
        }
        char[] chars = s.toCharArray();
        int p = 0, q = chars.length - 1;
        while (p < q) {
            if (!isVowel(chars[p])) {
                ++p;
                continue;
            }
```

```
            if (!isVowel(chars[q])) {
                --q;
                continue;
            }
            swap(chars, p++, q--);
        }
        return String.valueOf(chars);
    }

    private void swap(char[] chars, int i, int j) {
        char t = chars[i];
        chars[i] = chars[j];
        chars[j] = t;
    }

    private boolean isVowel(char c) {
        switch(c) {
        case 'a':
        case 'e':
        case 'i':
        case 'o':
        case 'u':
        case 'A':
        case 'E':
        case 'I':
        case 'O':
        case 'U':
            return true;
        default:
            return false;
        }
    }
}
```

**字符串转换整数 **

题目描述

请你来实现一个 atoi 函数，使其能将字符串转换成整数。

首先，该函数会根据需要丢弃无用的开头空格字符，直到寻找到第一个非空格的字符为止。

当我们寻找到的第一个非空字符为正或者负号时，则将该符号与之后面尽可能多的连续数字组合起来作为该整数的正负号；假如第一个非空字符是数字，则直接将其与之后连续的数字字符组合起来，形整数。

该字符串除了有效的整数部分之后也可能会存在多余的字符，这些字符可以被忽略，它们对于函数不该造成影响。

注意：假如该字符串中的第一个非空格字符不是一个有效整数字符、字符串为空或字符串仅包含空白符时，则你的函数不需要进行转换。

在任何情况下，若函数不能进行有效的转换时，请返回 0。

说明：

假设我们的环境只能存储 32 位大小的有符号整数，那么其数值范围为 [−231, 231 − 1]。如果数值过这个范围，请返回 INT_MAX (231 − 1) 或 INT_MIN (−231) 。

示例 1:

输入: "42"
输出: 42

示例 2:

输入: "   -42"
输出: -42
解释: 第一个非空白字符为 '-', 它是一个负号。
    我们尽可能将负号与后面所有连续出现的数字组合起来，最后得到 -42 。

示例 3:

输入: "4193 with words"
输出: 4193
解释: 转换截止于数字 '3' ，因为它的下一个字符不为数字。

示例 4:

输入: "words and 987"
输出: 0
解释: 第一个非空字符是 'w', 但它不是数字或正、负号。
    因此无法执行有效的转换。

示例 5:

输入: "-91283472332"
输出: -2147483648
解释: 数字 "-91283472332" 超过 32 位有符号整数范围。
    因此返回 INT_MIN (−231) 。

解法

遍历字符串，注意做溢出处理。

```
class Solution {
    public int myAtoi(String s) {
        if (s == null) return 0;
        int n = s.length();
        if (n == 0) return 0;
        int i = 0;
        while (s.charAt(i) == ' ') {
            // 仅包含空格
            if (++i == n) return 0;
        }
        int sign = 1;
        if (s.charAt(i) == '-') sign = -1;
        if (s.charAt(i) == '-' || s.charAt(i) == '+') ++i;
        int res = 0, flag = Integer.MAX_VALUE / 10;
```

```
        for (; i < n; ++i) {
            // 非数字，跳出循环体
            if (s.charAt(i) < '0' || s.charAt(i) > '9') break;
            // 溢出判断
            if (res > flag || (res == flag && s.charAt(i) > '7')) return sign > 0 ? Integer.MAX_VALUE :
Integer.MIN_VALUE;
            res = res * 10 + (s.charAt(i) - '0');
        }
        return sign * res;
    }
}
```

赎金信

题目描述

给定一个赎金信 (ransom) 字符串和一个杂志(magazine)字符串，判断第一个字符串ransom能不由第二个字符串magazines里面的字符构成。如果可以构成，返回 true ；否则返回 false。

(题目说明：为了不暴露赎金信字迹，要从杂志上搜索各个需要的字母，组成单词来表达意思。)

注意:

你可以假设两个字符串均只含有小写字母。

```
canConstruct("a", "b") -> false
canConstruct("aa", "ab") -> false
canConstruct("aa", "aab") -> true
```

解法

用一个数组或字典 chars 存放 magazine 中每个字母出现的次数。遍历 ransomNote 中每个字母判断 chars 是否包含即可。

```
class Solution {
    public boolean canConstruct(String ransomNote, String magazine) {
        int[] chars = new int[26];
        for (int i = 0, n = magazine.length(); i < n; ++i) {
            int idx = magazine.charAt(i) - 'a';
            ++chars[idx];
        }
        for (int i = 0, n = ransomNote.length(); i < n; ++i) {
            int idx = ransomNote.charAt(i) - 'a';
            if (chars[idx] == 0) return false;
            --chars[idx];
        }
        return true;
    }
}
```

两数相加

题目描述

**给出两个 非空 的链表用来表示两个非负的整数。其中，它们各自的位数是按照
序 的方式存储的，并且它们的每个节点只能存储 一位
数字。**

如果，我们将这两个数相加起来，则会返回一个新的链表来表示它们的和。

您可以假设除了数字 0 之外，这两个数都不会以 0 开头。

示例：

```
输入：(2 -> 4 -> 3) + (5 -> 6 -> 4)
输出：7 -> 0 -> 8
原因：342 + 465 = 807
```

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
class Solution {
    public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
        int carry = 0;
        ListNode dummy = new ListNode(-1);
        ListNode cur = dummy;
        while (l1 != null || l2 != null || carry != 0) {
            int t = (l1 == null ? 0 : l1.val) + (l2 == null ? 0 : l2.val) + carry;
            carry = t / 10;
            cur.next = new ListNode(t % 10);
            cur = cur.next;
            l1 = l1 == null ? null : l1.next;
            l2 = l2 == null ? null : l2.next;
        }
        return dummy.next;
    }
}
```

两数相加 II

题目描述

**给定两个非空链表来代表两个非负整数。数字最高位位于链表开始位置。它们的每个节点只
储单个数字。将这两数相加会返回一个新的链表。**

你可以假设除了数字 0 之外，这两个数字都不会以零开头。

进阶:

如果输入链表不能修改该如何处理？换句话说，你不能对列表中的节点进行翻转。

示例:

```
输入: (7 -> 2 -> 4 -> 3) + (5 -> 6 -> 4)
输出: 7 -> 8 -> 0 -> 7
```

解法

利用栈将数字逆序。

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
        Deque<Integer> s1 = new ArrayDeque<>();
        Deque<Integer> s2 = new ArrayDeque<>();
        for (; l1 != null; l1 = l1.next) {
            s1.push(l1.val);
        }
        for (; l2 != null; l2 = l2.next) {
            s2.push(l2.val);
        }
        int carry = 0;
        ListNode dummy = new ListNode(-1);
        while (!s1.isEmpty() || !s2.isEmpty() || carry != 0) {
            carry += (s1.isEmpty() ? 0 : s1.pop()) + (s2.isEmpty() ? 0 : s2.pop());
            // 创建结点，利用头插法将结点插入链表
            ListNode node = new ListNode(carry % 10);
            node.next = dummy.next;
            dummy.next = node;
            carry /= 10;
        }
        return dummy.next;
    }
}
```

从尾到头打印链表

题目描述

输入一个链表的头节点，从尾到头反过来返回每个节点的值（用数组返回）。

示例 1：

```
输入：head = [1,3,2]
输出：[2,3,1]
```

限制：

- 0 <= 链表长度 <= 10000

解法

栈实现。或者其它方式，见题解。

- **栈实现：**

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    public int[] reversePrint(ListNode head) {
        Stack<Integer> s = new Stack<>();
        while (head != null) {
            s.push(head.val);
            head = head.next;
        }
        int[] res = new int[s.size()];
        int i = 0;
        while (!s.isEmpty()) {
            res[i++] = s.pop();
        }
        return res;
    }
}
```

- **先计算链表长度 n，然后创建一个长度为 n 的结果数组。最后遍历链表，依次将节点值存放在数组（从后往前）。**

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    public int[] reversePrint(ListNode head) {
        if (head == null) return new int[]{};
        // 计算链表长度n
        int n = 0;
        ListNode cur = head;
        while (cur != null) {
            ++n;
            cur = cur.next;
        }
```

```
        int[] res = new int[n];
        cur = head;
        while (cur != null) {
            res[--n] = cur.val;
            cur = cur.next;
        }
        return res;
    }
}
```

删除链表的节点

给定单向链表的头指针和一个要删除的节点的值，定义一个函数删除该节点。

返回删除后的链表的头节点。

示例 1:

```
输入: head = [4,5,1,9], val = 5
输出: [4,1,9]
解释: 给定你链表中值为 5 的第二个节点，那么在调用了你的函数之后，该链表应变为 4 -> 1 -> 9.
```

示例 2:

```
输入: head = [4,5,1,9], val = 1
输出: [4,5,9]
解释: 给定你链表中值为 1 的第三个节点，那么在调用了你的函数之后，该链表应变为 4 -> 5 -> 9.
```

说明:

- **题目保证链表中节点的值互不相同**
- **若使用 C 或 C++ 语言，你不需要 free 或 delete 被删除的节点**

解法

定义一个虚拟头节点 dummy 指向 head，pre** 指针初始指向 dummy。**

循环遍历链表，pre** 往后移动。当指针 pre.next 指向的节点的值等于 val 时退出循环，将 pre.next 指向 pre.next.next，然后返回 dummy.next。**

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    public ListNode deleteNode(ListNode head, int val) {
        ListNode dummy = new ListNode(0);
        dummy.next = head;
        ListNode pre = dummy;
        while (pre.next != null && pre.next.val != val) {
```

```
            pre = pre.next;
        }
        pre.next = pre.next == null ? null : pre.next.next;
        return dummy.next;
    }
}
```

删除排序链表中的重复元素

题目描述

给定一个排序链表，删除所有重复的元素，使得每个元素只出现一次。

示例 1:

```
输入: 1->1->2
输出: 1->2
```

示例 2:

```
输入: 1->1->2->3->3
输出: 1->2->3
```

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
class Solution {
    public ListNode deleteDuplicates(ListNode head) {
        ListNode cur = head;
        while (cur != null && cur.next != null) {
            if (cur.val == cur.next.val) {
                cur.next = cur.next.next;
            } else {
                cur = cur.next;
            }
        }
        return head;
    }
}
```

删除排序链表中的重复元素 II

题目描述

给定一个排序链表，删除所有含有重复数字的节点，只保留原始链表中 没有重复出现的数字。

示例 1:

```
输入: 1->2->3->3->4->4->5
输出: 1->2->5
```

示例 2:

```
输入: 1->1->1->2->3
输出: 2->3
```

解法

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
class Solution {
    public ListNode deleteDuplicates(ListNode head) {
        ListNode dummy = new ListNode(-1, head);
        ListNode cur = dummy;
        while (cur.next != null && cur.next.next != null) {
            if (cur.next.val == cur.next.next.val) {
                int val = cur.next.val;
                while (cur.next != null && cur.next.val == val) {
                    cur.next = cur.next.next;
                }
            } else {
                cur = cur.next;
            }
        }
        return dummy.next;
    }
}
```

移除链表元素

题目描述

删除链表中等于给定值 val 的所有节点。

示例:

```
输入: 1->2->6->3->4->5->6, val = 6
输出: 1->2->3->4->5
```

解法

```
/**
```

```
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
class Solution {
    public ListNode removeElements(ListNode head, int val) {
        ListNode dummy = new ListNode(-1, head);
        ListNode pre = dummy;
        while (pre != null && pre.next != null) {
            if (pre.next.val != val) pre = pre.next;
            else pre.next = pre.next.next;
        }
        return dummy.next;
    }
}
```

两两交换链表中的节点

题目描述

给定一个链表，两两交换其中相邻的节点，并返回交换后的链表。

你不能只是单纯的改变节点内部的值，而是需要实际的进行节点交换。**

示例:

```
1->2->3->4
```

解法

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
class Solution {
    public ListNode swapPairs(ListNode head) {
        ListNode dummy = new ListNode(0, head);
        ListNode pre = dummy, cur = head;
        while (cur != null && cur.next != null) {
            ListNode t = cur.next;
            cur.next = t.next;
```

```
            t.next = cur;
            pre.next = t;
            pre = cur;
            cur = pre.next;

        }
        return dummy.next;
    }
}
```

排序链表

题目描述

在 O(n log n) 时间复杂度和常数级空间复杂度下，对链表进行排序。

示例 1:

输入: 4->2->1->3
输出: 1->2->3->4

示例 2:

输入: -1->5->3->4->0
输出: -1->0->3->4->5

解法

先用快慢指针找到链表中点，然后分成左右两个链表，递归排序左右链表。最后合并两个排序的链表可。

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
class Solution {
    public ListNode sortList(ListNode head) {
        if (head == null || head.next == null) {
            return head;
        }
        ListNode slow = head, fast = head.next;
        while (fast != null && fast.next != null) {
            slow = slow.next;
            fast = fast.next.next;
        }
        ListNode t = slow.next;
        slow.next = null;
        ListNode l1 = sortList(head);
```

```
        ListNode l2 = sortList(t);
        ListNode dummy = new ListNode(0);
        ListNode cur = dummy;
        while (l1 != null && l2 != null) {
            if (l1.val <= l2.val) {
                cur.next = l1;
                l1 = l1.next;
            } else {
                cur.next = l2;
                l2 = l2.next;
            }
            cur = cur.next;
        }
        cur.next = l1 == null ? l2 : l1;
        return dummy.next;
    }
}
```

反转链表

题目描述

反转一个单链表。

示例:

```
输入: 1->2->3->4->5->NULL
输出: 5->4->3->2->1->NULL
```

进阶:**

****你可以迭代或递归地反转链表。你能否用两种方法解决这道题?**

解法

定义指针 p、q 分别指向头节点和下一个节点，pre 指向头节点的前一个节点。****

遍历链表，改变指针 p 指向的节点的指向，将其指向 pre 指针指向的节点，即 p.next = pre。然后 re 指针指向 p，p、q 指针往前走。****

当遍历结束后，返回 pre 指针即可。

迭代版本

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    public ListNode reverseList(ListNode head) {
        ListNode pre = null, p = head;
```

```
        while (p != null) {
            ListNode q = p.next;
            p.next = pre;
            pre = p;
            p = q;
        }
        return pre;
    }
}
```

递归版本

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    public ListNode reverseList(ListNode head) {
        if (head == null || head.next == null) {
            return head;
        }
        ListNode res = reverseList(head.next);
        head.next.next = head;
        head.next = null;
        return res;
    }
}
```

二叉树的前序遍历

题目描述

给定一个二叉树，返回它的 前序 遍历。

** **示例:**

```
输入: [1,null,2,3]
   1
    \
     2
    /
   3

输出: [1,2,3]
```

进阶:** 递归算法很简单，你可以通过迭代算法完成吗？ **

解法

递归遍历或利用栈实现非递归遍历。

非递归的思路如下：

1. **定义一个栈，先将根节点压入栈**
2. **若栈不为空，每次从栈中弹出一个节点**
3. **处理该节点**
4. **先把节点右孩子压入栈，接着把节点左孩子压入栈（如果有孩子节点）**
5. **重复 2-4**
6. **返回结果**

递归

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution {

    private List<Integer> res;

    public List<Integer> preorderTraversal(TreeNode root) {
        res = new ArrayList<>();
        preorder(root);
        return res;
    }

    private void preorder(TreeNode root) {
        if (root != null) {
            res.add(root.val);
            preorder(root.left);
            preorder(root.right);
        }
    }
}
```

非递归

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
```

```
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution {
    public List<Integer> preorderTraversal(TreeNode root) {
        if (root == null) {
            return Collections.emptyList();
        }
        List<Integer> res = new ArrayList<>();
        Deque<TreeNode> s = new ArrayDeque<>();
        s.push(root);
        while (!s.isEmpty()) {
            TreeNode node = s.pop();
            res.add(node.val);
            if (node.right != null) {
                s.push(node.right);
            }
            if (node.left != null) {
                s.push(node.left);
            }
        }
        return res;
    }
}
```

二叉树的后序遍历

题目描述

给定一个二叉树，返回它的 后序 遍历。

示例:

```
输入: [1,null,2,3]
   1
    \
     2
    /
   3

输出: [3,2,1]
```

进阶:** 递归算法很简单，你可以通过迭代算法完成吗？**

解法

递归遍历或利用栈实现非递归遍历。

非递归的思路如下：

先序遍历的顺序是：头、左、右，如果我们改变左右孩子的顺序，就能将顺序变成：头、右、左。

我们先不打印头节点，而是存放到另一个收集栈 s2 中，最后遍历结束，输出收集栈元素，即是后序历：左、右、头。

递归

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution {

    private List<Integer> res;

    public List<Integer> postorderTraversal(TreeNode root) {
        res = new ArrayList<>();
        postorder(root);
        return res;
    }

    private void postorder(TreeNode root) {
        if (root != null) {
            postorder(root.left);
            postorder(root.right);
            res.add(root.val);
        }
    }
}
```

非递归

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
```

```
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution {
    public List<Integer> postorderTraversal(TreeNode root) {
        if (root == null) {
            return Collections.emptyList();
        }
        Deque<TreeNode> s1 = new ArrayDeque<>();
        List<Integer> s2 = new ArrayList<>();
        s1.push(root);
        while (!s1.isEmpty()) {
            TreeNode node = s1.pop();
            s2.add(node.val);
            if (node.left != null) {
                s1.push(node.left);
            }
            if (node.right != null) {
                s1.push(node.right);
            }
        }
        Collections.reverse(s2);
        return s2;
    }
}
```

二叉树的中序遍历

题目描述

给定一个二叉树，返回它的中序 ** 遍历。**

示例:

```
输入: [1,null,2,3]
   1
    \
     2
    /
   3

输出: [1,3,2]
```

进阶:** 递归算法很简单，你可以通过迭代算法完成吗？**

解法

递归遍历或利用栈实现非递归遍历。

非递归的思路如下：

1. **定义一个栈**

2. **将树的左节点依次入栈**

3. **左节点为空时，弹出栈顶元素并处理**

4. **重复 2-3 的操作**

递归

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution {

    private List<Integer> res;

    public List<Integer> inorderTraversal(TreeNode root) {
        res = new ArrayList<>();
        inorder(root);
        return res;
    }

    private void inorder(TreeNode root) {
        if (root != null) {
            inorder(root.left);
            res.add(root.val);
            inorder(root.right);
        }
    }
}
```

非递归

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
```

```
 *         this.right = right;
 *     }
 * }
 */
class Solution {
    public List<Integer> inorderTraversal(TreeNode root) {
        if (root == null) {
            return Collections.emptyList();
        }
        List<Integer> res = new ArrayList<>();
        Deque<TreeNode> s = new ArrayDeque<>();
        while (root != null || !s.isEmpty()) {
            if (root != null) {
                s.push(root);
                root = root.left;
            } else {
                root = s.pop();
                res.add(root.val);
                root = root.right;
            }
        }
        return res;
    }
}
```

最小栈

题目描述

设计一个支持 push，pop，top 操作，并能在常数时间内检索到最小元素的栈。

- **push(x) -- 将元素 x 推入栈中。**
- **pop() -- 删除栈顶的元素。**
- **top() -- 获取栈顶元素。**
- **getMin() -- 检索栈中的最小元素。**

示例:

```
MinStack minStack = new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.getMin();   --> 返回 -3.
minStack.pop();
minStack.top();      --> 返回 0.
minStack.getMin();   --> 返回 -2.
```

解法

```
class MinStack {

    private Deque<Integer> s;
```

```
    private Deque<Integer> helper;

    /** initialize your data structure here. */
    public MinStack() {
        s = new ArrayDeque<>();
        helper = new ArrayDeque<>();
    }

    public void push(int x) {
        s.push(x);
        int element = helper.isEmpty() || x < helper.peek() ? x : helper.peek();
        helper.push(element);
    }

    public void pop() {
        s.pop();
        helper.pop();
    }

    public int top() {
        return s.peek();
    }

    public int getMin() {
        return helper.peek();
    }
}

/**
 * Your MinStack object will be instantiated and called as such:
 * MinStack obj = new MinStack();
 * obj.push(x);
 * obj.pop();
 * int param_3 = obj.top();
 * int param_4 = obj.getMin();
 */
```

队列的最大值

题目描述

请定义一个队列并实现函数 max_value 得到队列里的最大值，要求函数max_value、push_back** 和 **op_front** 的**均摊**时间复杂度都是 O(1)。**

若队列为空，pop_front** 和 **max_value** 需要返回 -1**

示例 1:

```
输入:
["MaxQueue","push_back","push_back","max_value","pop_front","max_value"]
[[],[1],[2],[],[],[]]
输出: [null,null,null,2,1,2]
```

示例 2:

输入:
["MaxQueue","pop_front","max_value"]
[[],[],[]]
输出: [null,-1,-1]

限制:

- 1 <= push_back,pop_front,max_value的总操作数 <= 10000
- 1 <= value <= 10^5

解法

利用一个辅助队列按单调顺序存储当前队列的最大值。

```
class MaxQueue {
    private Deque<Integer> p;
    private Deque<Integer> q;

    public MaxQueue() {
        p = new ArrayDeque<>();
        q = new ArrayDeque<>();
    }

    public int max_value() {
        return q.isEmpty() ? -1 : q.peekFirst();
    }

    public void push_back(int value) {
        while (!q.isEmpty() && q.peekLast() < value) {
            q.pollLast();
        }
        p.offerLast(value);
        q.offerLast(value);
    }

    public int pop_front() {
        if (p.isEmpty()) return -1;
        int res = p.pollFirst();
        if (q.peek() == res) q.pollFirst();
        return res;
    }
}

/**
 * Your MaxQueue object will be instantiated and called as such:
 * MaxQueue obj = new MaxQueue();
 * int param_1 = obj.max_value();
 * obj.push_back(value);
 * int param_3 = obj.pop_front();
 */
```

冒泡排序

冒泡排序是最简单的排序之一了，其大体思想就是通过与相邻元素的比较和交换来把小的数交换到最面。这个过程类似于水泡向上升一样，因此而得名。举个栗子，对5,3,8,6,4这个无序序列进行冒泡排。首先从后向前冒泡，4和6比较，把4交换到前面，序列变成5,3,8,4,6。同理4和8交换，变成5,3,4,86,3和4无需交换。5和3交换，变成3,5,4,8,6,3.这样一次冒泡就完了，把最小的数3排到最前面了。剩下的序列依次冒泡就会得到一个有序序列。冒泡排序的时间复杂度为O(n^2)。

实现代码：

```
public class BubbleSort {

    public static void bubbleSort(int[] arr) {
        if(arr == null || arr.length == 0)
            return ;
        for(int i=0; i<arr.length-1; i++) {
            for(int j=arr.length-1; j>i; j--) {
                if(arr[j] < arr[j-1]) {
                    swap(arr, j-1, j);
                }
            }
        }
    }


    public static void swap(int[] arr, int i, int j) {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }
}
```

选择排序

选择排序的思想其实和冒泡排序有点类似，都是在一次排序后把最小的元素放到最前面。但是过程不，冒泡排序是通过相邻的比较和交换。而选择排序是通过对整体的选择。举个栗子，对5,3,8,6,4这个序序列进行简单选择排序，首先要选择5以外的最小数来和5交换，也就是选择3和5交换，一次排序就变成了3,5,8,6,4.对剩下的序列一次进行选择和交换，最终就会得到一个有序序列。其实选择排序以看成冒泡排序的优化，因为其目的相同，只是选择排序只有在确定了最小数的前提下才进行交换，大减少了交换的次数。选择排序的时间复杂度为O(n^2)

实现代码：

```
public class SelectSort {

    public static void selectSort(int[] arr) {
        if(arr == null || arr.length == 0)
            return ;
        int minIndex = 0;
        for(int i=0; i<arr.length-1; i++) { //只需要比较n-1次
            minIndex = i;
            for(int j=i+1; j<arr.length; j++) { //从i+1开始比较，因为minIndex默认为i了，i就没必要
了。
                if(arr[j] < arr[minIndex]) {
                    minIndex = j;
                }
```

```
            }

            if(minIndex != i) { //如果minIndex不为i，说明找到了更小的值，交换之。
                swap(arr, i, minIndex);
            }
        }

    }

    public static void swap(int[] arr, int i, int j) {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }

}
```

插入排序

插入排序不是通过交换位置而是通过比较找到合适的位置插入元素来达到排序的目的的。相信大家都过打扑克牌的经历，特别是牌数较大的。在分牌时可能要整理自己的牌，牌多的时候怎么整理呢？就拿到一张牌，找到一个合适的位置插入。这个原理其实和插入排序是一样的。举个栗子，对5,3,8,6,4个无序序列进行简单插入排序，首先假设第一个数的位置时正确的，想一下在拿到第一张牌的时候，必要整理。然后3要插到5前面，把5后移一位，变成3,5,8,6,4.想一下整理牌的时候应该也是这样吧。后8不用动，6插在8前面，8后移一位，4插在5前面，从5开始都向后移一位。注意在插入一个数的时要保证这个数前面的数已经有序。简单插入排序的时间复杂度也是O(n^2)。

实现代码：

```
public class InsertSort {

    public static void insertSort(int[] arr) {
        if(arr == null || arr.length == 0)
            return ;

        for(int i=1; i<arr.length; i++) { //假设第一个数位置时正确的；要往后移，必须要假设第一个。

            int j = i;
            int target = arr[i]; //待插入的

            //后移
            while(j > 0 && target < arr[j-1]) {
                arr[j] = arr[j-1];
                j --;
            }

            //插入
            arr[j] = target;
        }

    }

}
```

快速排序

快速排序一听名字就觉得很高端，在实际应用当中快速排序确实也是表现最好的排序算法。快速排序然高端，但其实其思想是来自冒泡排序，冒泡排序是通过相邻元素的比较和交换把最小的冒泡到最顶，而快速排序是比较和交换小数和大数，这样一来不仅把小数冒泡到上面同时也把大数沉到下面。

举个栗子：对5,3,8,6,4这个无序序列进行快速排序，思路是右指针找比基准数小的，左指针找比基准大的，交换之。

5,3,8,6,4 用5作为比较的基准，最终会把5小的移动到5的左边，比5大的移动到5的右边。

5,3,8,6,4 首先设置i,j两个指针分别指向两端，j指针先扫描（思考一下为什么？）4比5小停止。然后i描，8比5大停止。交换i,j位置。

5,3,4,6,8 然后j指针再扫描，这时j扫描4时两指针相遇。停止。然后交换4和基准数。

4,3,5,6,8 一次划分后达到了左边比5小，右边比5大的目的。之后对左右子序列递归排序，最终得到序序列。

上面留下来了一个问题为什么一定要j指针先动呢？首先这也不是绝对的，这取决于基准数的位置，因在最后两个指针相遇的时候，要交换基准数到相遇的位置。一般选取第一个数作为基准数，那么就是左边，所以最后相遇的数要和基准数交换，那么相遇的数一定要比基准数小。所以j指针先移动才能先到比基准数小的数。

快速排序是不稳定的，其时间平均时间复杂度是O(nlgn)。

实现代码：

```
public class QuickSort {
    //一次划分
    public static int partition(int[] arr, int left, int right) {
        int pivotKey = arr[left];
        int pivotPointer = left;

        while(left < right) {
            while(left < right && arr[right] >= pivotKey)
                right --;
            while(left < right && arr[left] <= pivotKey)
                left ++;
            swap(arr, left, right); //把大的交换到右边，把小的交换到左边。
        }
        swap(arr, pivotPointer, left); //最后把pivot交换到中间
        return left;
    }

    public static void quickSort(int[] arr, int left, int right) {
        if(left >= right)
            return ;
        int pivotPos = partition(arr, left, right);
        quickSort(arr, left, pivotPos-1);
        quickSort(arr, pivotPos+1, right);
    }

    public static void sort(int[] arr) {
```

```
        if(arr == null || arr.length == 0)
            return ;
        quickSort(arr, 0, arr.length-1);
    }

    public static void swap(int[] arr, int left, int right) {
        int temp = arr[left];
        arr[left] = arr[right];
        arr[right] = temp;
    }

}
```

其实上面的代码还可以再优化，上面代码中基准数已经在pivotKey中保存了，所以不需要每次交换设置一个temp变量，在交换左右指针的时候只需要先后覆盖就可以了。这样既能减少空间的使用还降低赋值运算的次数。优化代码如下：

```
public class QuickSort {

    /**
     * 划分
     * @param arr
     * @param left
     * @param right
     * @return
     */
    public static int partition(int[] arr, int left, int right) {
        int pivotKey = arr[left];

        while(left < right) {
            while(left < right && arr[right] >= pivotKey)
                right --;
            arr[left] = arr[right]; //把小的移动到左边
            while(left < right && arr[left] <= pivotKey)
                left ++;
            arr[right] = arr[left]; //把大的移动到右边
        }
        arr[left] = pivotKey; //最后把pivot赋值到中间
        return left;
    }

    /**
     * 递归划分子序列
     * @param arr
     * @param left
     * @param right
     */
    public static void quickSort(int[] arr, int left, int right) {
        if(left >= right)
            return ;
        int pivotPos = partition(arr, left, right);
        quickSort(arr, left, pivotPos-1);
        quickSort(arr, pivotPos+1, right);
    }
```

```
    public static void sort(int[] arr) {
        if(arr == null || arr.length == 0)
            return ;
        quickSort(arr, 0, arr.length-1);
    }

}
```

总结快速排序的思想：冒泡+二分+递归分治，慢慢体会。。。

堆排序

堆排序是借助堆来实现的选择排序，思想同简单的选择排序，以下以大顶堆为例。注意：如果想升序序就使用大顶堆，反之使用小顶堆。原因是堆顶元素需要交换到序列尾部。

首先，实现堆排序需要解决两个问题：

如何由一个无序序列键成一个堆？

如何在输出堆顶元素之后，调整剩余元素成为一个新的堆？

第一个问题，可以直接使用线性数组来表示一个堆，由初始的无序序列建成一个堆就需要自底向上从一个非叶元素开始挨个调整成一个堆。

第二个问题，怎么调整成堆？首先是将堆顶元素和最后一个元素交换。然后比较当前堆顶元素的左右子节点，因为除了当前的堆顶元素，左右孩子堆均满足条件，这时需要选择当前堆顶元素与左右孩子点的较大者（大顶堆）交换，直至叶子节点。我们称这个自堆顶自叶子的调整成为筛选。

从一个无序序列建堆的过程就是一个反复筛选的过程。若将此序列看成是一个完全二叉树，则最后一非终端节点是n/2取底个元素，由此筛选即可。举个栗子：

**49,38,65,97,76,13,27,49序列的堆排序建初始堆和调整的过程如下 **

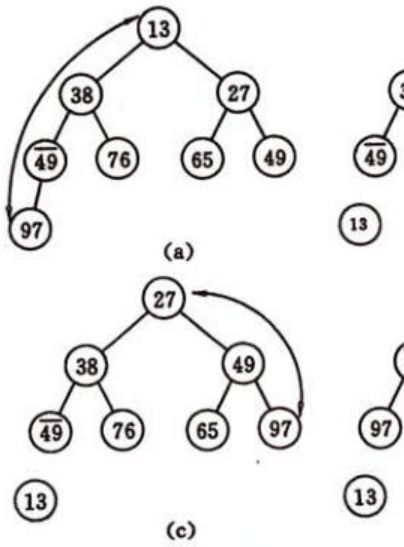


图 10.11　输出堆顶元素并调
(a) 堆；　(b) 13 和 97 交换后的情形；
(d) 27 和 97 交换后再进行调

原文链接：什么是数据结构?

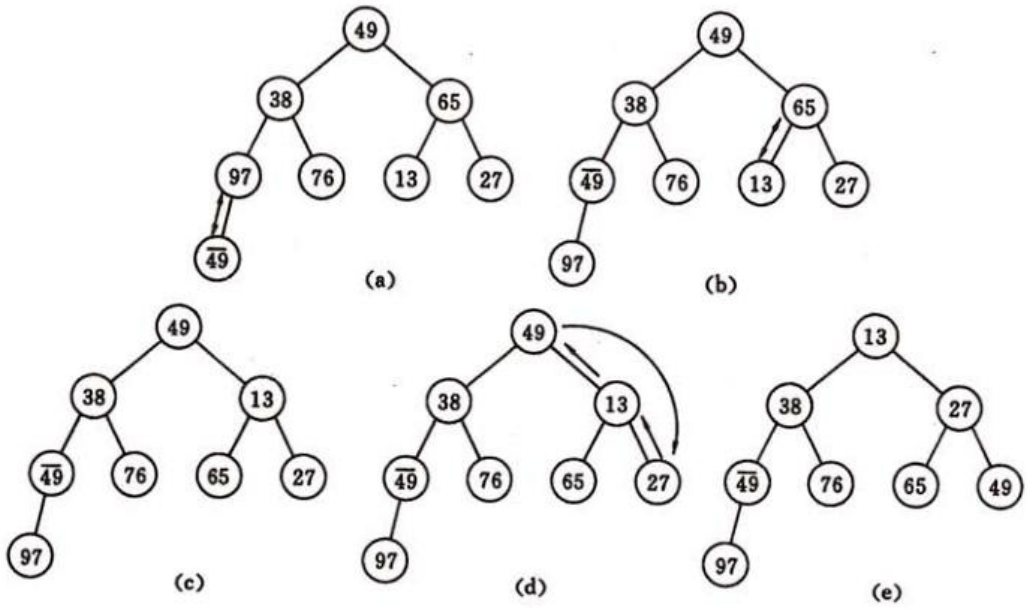


图 10.12　建初始堆过程示例

(a) 无序序列；(b) 97 被筛选之后的状态；(c) 65 被筛选之后的状态；

(d) 38 被筛选之后的状态；(e) 49 被筛选之后建成的堆

实现代码：

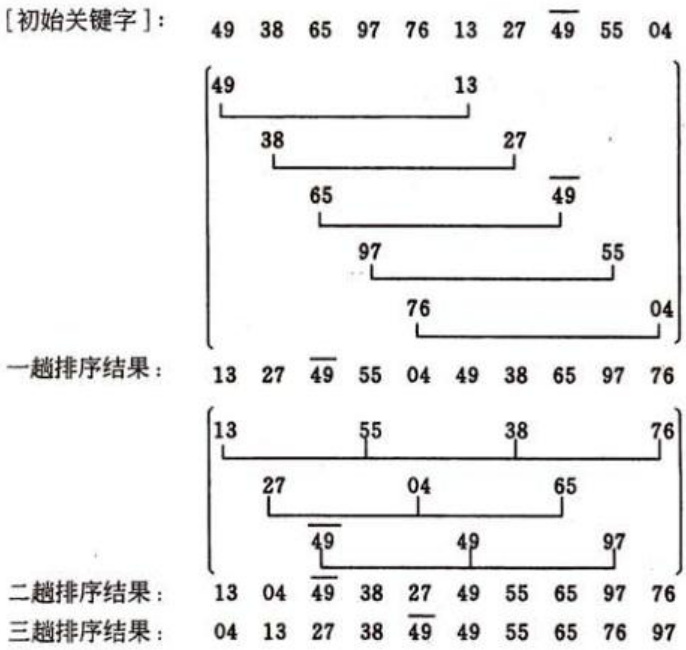
```
public class HeapSort {

    /**
     * 堆筛选，除了start之外，start~end均满足大顶堆的定义。
     * 调整之后start~end称为一个大顶堆。
     * @param arr 待调整数组
     * @param start 起始指针
     * @param end 结束指针
     */
    public static void heapAdjust(int[] arr, int start, int end) {
        int temp = arr[start];

        for(int i=2*start+1; i<=end; i*=2) {
            //左右孩子的节点分别为2*i+1,2*i+2

            //选择出左右孩子较小的下标
            if(i < end && arr[i] < arr[i+1]) {
                i ++;
            }
            if(temp >= arr[i]) {
                break; //已经为大顶堆，=保持稳定性。
            }
            arr[start] = arr[i]; //将子节点上移
            start = i; //下一轮筛选
        }

        arr[start] = temp; //插入正确的位置
    }
```

```
    public static void heapSort(int[] arr) {
        if(arr == null || arr.length == 0)
            return ;

        //建立大顶堆
        for(int i=arr.length/2; i>=0; i--) {
            heapAdjust(arr, i, arr.length-1);
        }

        for(int i=arr.length-1; i>=0; i--) {
            swap(arr, 0, i);
            heapAdjust(arr, 0, i-1);
        }

    }

    public static void swap(int[] arr, int i, int j) {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }
}
```

希尔排序

希尔排序是插入排序的一种高效率的实现，也叫缩小增量排序。简单的插入排序中，如果待排序列是序时，时间复杂度是O(n)，如果序列是基本有序的，使用直接插入排序效率就非常高。希尔排序就利了这个特点。基本思想是：先将整个待排记录序列分割成为若干子序列分别进行直接插入排序，待整序列中的记录基本有序时再对全体记录进行一次直接插入排序。

举个栗子：



从上述排序过程可见，希尔排序的特点是，子序列的构成不是简单的逐段分割，而是将某个相隔某个

量的记录组成一个子序列。如上面的例子，第一堂排序时的增量为5，第二趟排序的增量为3。由于前趟的插入排序中记录的关键字是和同一子序列中的前一个记录的关键字进行比较，因此关键字较小的录就不是一步一步地向前挪动，而是跳跃式地往前移，从而使得进行最后一趟排序时，整个序列已经到基本有序，只要作记录的少量比较和移动即可。因此希尔排序的效率要比直接插入排序高。

希尔排序的分析是复杂的，时间复杂度是所取增量的函数，这涉及一些数学上的难题。但是在大量实的基础上推出当n在某个范围内时，时间复杂度可以达到O(n^1.3)。

实现代码：

```
public class ShellSort {

    /**
     * 希尔排序的一趟插入
     * @param arr 待排数组
     * @param d 增量
     */
    public static void shellInsert(int[] arr, int d) {
        for(int i=d; i<arr.length; i++) {
            int j = i - d;
            int temp = arr[i];    //记录要插入的数据
            while (j>=0 && arr[j]>temp) {  //从后向前，找到比其小的数的位置
                arr[j+d] = arr[j];    //向后挪动
                j -= d;
            }

            if (j != i - d)    //存在比其小的数
                arr[j+d] = temp;

        }
    }

    public static void shellSort(int[] arr) {
        if(arr == null || arr.length == 0)
            return ;
        int d = arr.length / 2;
        while(d >= 1) {
            shellInsert(arr, d);
            d /= 2;
        }
    }

}
```

归并排序

归并排序是另一种不同的排序方法，因为归并排序使用了递归分治的思想，所以理解起来比较容易。基本思想是，先递归划分子问题，然后合并结果。把待排序列看成由两个有序的子序列，然后合并两子序列，然后把子序列看成由两个有序序列。。。。。倒着来看，其实就是先两两合并，然后四四合。。。最终形成有序序列。空间复杂度为O(n)，时间复杂度为O(nlogn)。

**举个栗子： **

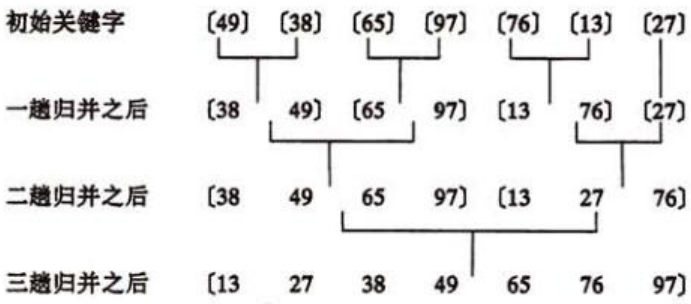


图 10.13　2-路归并排序示例

实现代码：

```
public class MergeSort {

    public static void mergeSort(int[] arr) {
        mSort(arr, 0, arr.length-1);
    }

    /**
     * 递归分治
     * @param arr 待排数组
     * @param left 左指针
     * @param right 右指针
     */
    public static void mSort(int[] arr, int left, int right) {
        if(left >= right)
            return ;
        int mid = (left + right) / 2;

        mSort(arr, left, mid); //递归排序左边
        mSort(arr, mid+1, right); //递归排序右边
        merge(arr, left, mid, right); //合并
    }

    /**
     * 合并两个有序数组
     * @param arr 待合并数组
     * @param left 左指针
     * @param mid 中间指针
     * @param right 右指针
     */
    public static void merge(int[] arr, int left, int mid, int right) {
        //[left, mid] [mid+1, right]
        int[] temp = new int[right - left + 1]; //中间数组

        int i = left;
        int j = mid + 1;
        int k = 0;
        while(i <= mid && j <= right) {
            if(arr[i] <= arr[j]) {
```

```
                temp[k++] = arr[i++];
            }
            else {
                temp[k++] = arr[j++];
            }
        }

        while(i <= mid) {
            temp[k++] = arr[i++];
        }

        while(j <= right) {
            temp[k++] = arr[j++];
        }

        for(int p=0; p<temp.length; p++) {
            arr[left + p] = temp[p];
        }

    }
}
```

计数排序

如果在面试中有面试官要求你写一个O(n)时间复杂度的排序算法，你千万不要立刻说：这不可能！虽前面基于比较的排序的下限是O(nlogn)。但是确实也有线性时间复杂度的排序，只不过有前提条件就是待排序的数要满足一定的范围的整数，而且计数排序需要比较多的辅助空间。其基本思想是，用排序的数作为计数数组的下标，统计每个数字的个数。然后依次输出即可得到有序序列。

实现代码：

```
public class CountSort {

    public static void countSort(int[] arr) {
        if(arr == null || arr.length == 0)
            return ;

        int max = max(arr);

        int[] count = new int[max+1];
        Arrays.fill(count, 0);

        for(int i=0; i<arr.length; i++) {
            count[arr[i]] ++;
        }

        int k = 0;
        for(int i=0; i<=max; i++) {
            for(int j=0; j<count[i]; j++) {
                arr[k++] = i;
            }
        }

    }
```

```
    public static int max(int[] arr) {
        int max = Integer.MIN_VALUE;
        for(int ele : arr) {
            if(ele > max)
                max = ele;
        }

        return max;
    }

}
```

桶排序

桶排序算是计数排序的一种改进和推广，但是网上有许多资料把计数排序和桶排序混为一谈。其实桶序要比计数排序复杂许多。

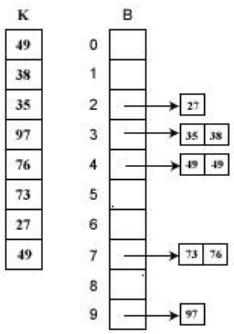
对桶排序的分析和解释借鉴这位兄弟的文章（有改动）： http://hxraid.iteye.com/blog/647759

桶排序的基本思想：

假设有一组长度为N的待排关键字序列K[1....n]。首先将这个序列划分成M个的子区间(桶) 。然后基某种映射函数 ， 将待排序列的关键字k映射到第i个桶中(即桶数组B的下标 i) ， 那么该关键字k就作为B i]中的元素(每个桶B[i]都是一组大小为N/M的序列)。接着对每个桶B[i]中的所有元素进行比较排序(以使用快排)。然后依次枚举输出B[0]....B[M]中的全部内容即是一个有序序列。bindex=f(key)其中bindex 为桶数组B的下标(即第bindex个桶), k为待排序列的关键字。桶排序之所以能够高效，其关在于这个映射函数，它必须做到：如果关键字k1<k2，那么f(k1)<=f(k2)。

也就是说B(i)中的最小数据都要大于B(i-1)中最大数据。很显然，映射函数的确定与数据本身的特点很大的关系。

**举个栗子： **



**假如待排序列K={49、38、35、97、76、73、27、49}。这些数据全部在1—100之间。因此我们制10个桶，然后确定映射函数f(k)=k/10。则第一个关键字49将定位到第4个桶中(49/10=4)。依次将有关键字全部堆入桶中，并在每个非空的桶中进行快速排序后得到如图所示。只要顺序输出每个B[i]的数据就可以得到有序序列了。 **

桶排序分析：

**桶排序利用函数的映射关系，减少了几乎所有的比较工作。实际上，桶排序的f(k)值的计算，其作就相当于快排中划分，希尔排序中的子序列，归并排序中的子问题，已经把大量数据分割成了基本有的数据块(桶)。然后只需要对桶中的少量数据做先进的比较排序即可。 **

对N个关键字进行桶排序的时间复杂度分为两个部分：

(1) 循环计算每个关键字的桶映射函数，这个时间复杂度是O(N)。

(2) 利用先进的比较排序算法对每个桶内的所有数据进行排序，其时间复杂度为∑ O(Ni*logNi) 。其Ni 为第i个桶的数据量。

很显然，第(2)部分是桶排序性能好坏的决定因素。尽量减少桶内数据的数量是提高效率的唯一办法(为基于比较排序的最好平均时间复杂度只能达到O(N*logN)了)。因此，我们需要尽量做到下面两点：

(1) 映射函数f(k)能够将N个数据平均的分配到M个桶中，这样每个桶就有[N/M]个数据量。

(2) 尽量的增大桶的数量。极限情况下每个桶只能得到一个数据，这样就完全避开了桶内数据的“比”排序操作。当然，做到这一点很不容易，数据量巨大的情况下，f(k)函数会使得桶集合的数量巨大空间浪费严重。这就是一个时间代价和空间代价的权衡问题了。

对于N个待排数据，M个桶，平均每个桶[N/M]个数据的桶排序平均时间复杂度为：

O(N)+O(M***(N/M)**log(N/M))=O(N+N**(logN-logM))=O(N+N*****logN-N*logM) 当N=M时即极限情况下每个桶只有一个数据时。桶排序的最好效率能够达到O(N)。**

总结： 桶排序的平均时间复杂度为线性的O(N+C)，其中C=N(logN-logM)。如果相对于同样的N，桶量M越大，其效率越高，最好的时间复杂度达到O(N)。 当然桶排序的空间复杂度为O(N+M)，如果输数据非常庞大，而桶的数量也非常多，则空间代价无疑是昂贵的。此外，桶排序是稳定的。*

实现代码：

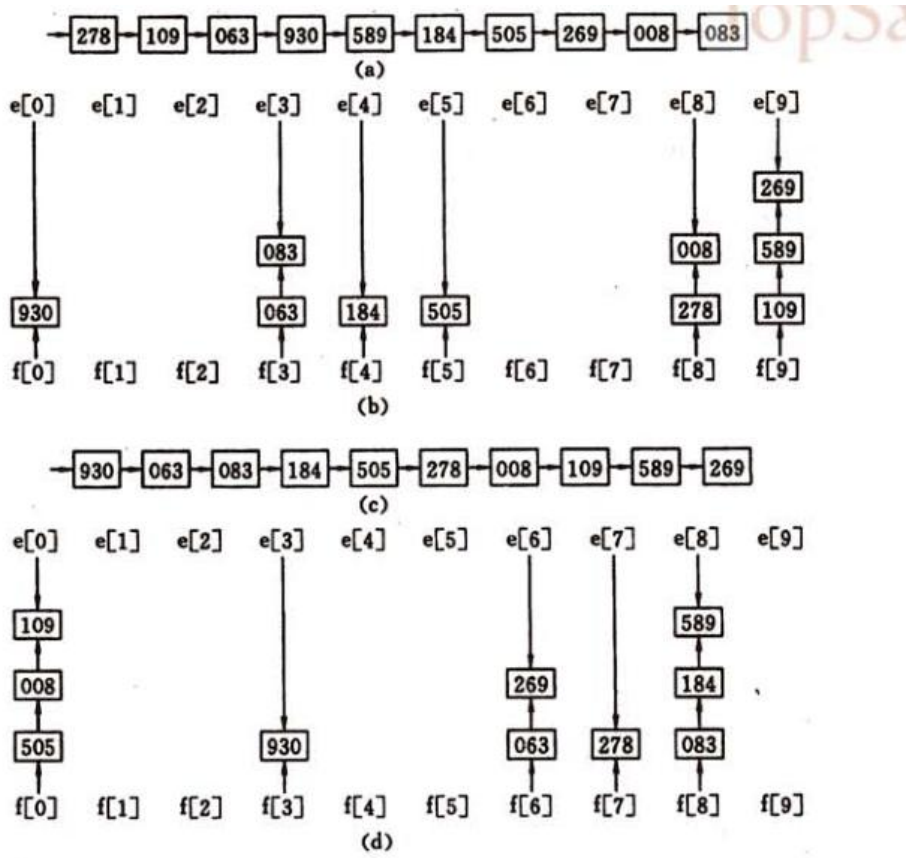
```
public class BucketSort {

    public static void bucketSort(int[] arr) {
        if(arr == null && arr.length == 0)
            return ;

        int bucketNums = 10; //这里默认为10，规定待排数[0,100)
        List<List<Integer>> buckets = new ArrayList<List<Integer>>(); //桶的索引

        for(int i=0; i<10; i++) {
            buckets.add(new LinkedList<Integer>()); //用链表比较合适
        }

        //划分桶
        for(int i=0; i<arr.length; i++) {
            buckets.get(f(arr[i])).add(arr[i]);
        }

        //对每个桶进行排序
        for(int i=0; i<buckets.size(); i++) {
            if(!buckets.get(i).isEmpty()) {
                Collections.sort(buckets.get(i)); //对每个桶进行快排
            }
        }

        //还原排好序的数组
        int k = 0;
        for(List<Integer> bucket : buckets) {
```

```
                for(int ele : bucket) {
                    arr[k++] = ele;
                }
            }
        }

        /**
         * 映射函数
         * @param x
         * @return
         */
        public static int f(int x) {
            return x / 10;
        }

}
```

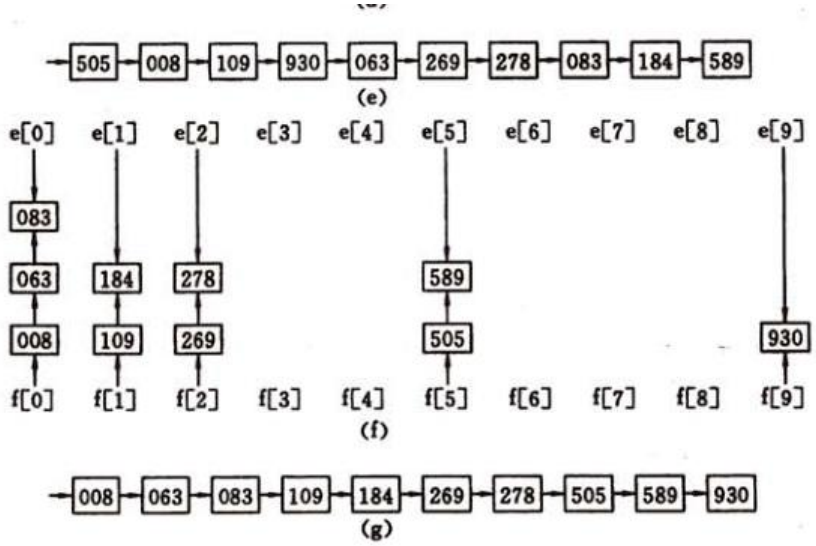
基数排序

基数排序又是一种和前面排序方式不同的排序方式，基数排序不需要进行记录关键字之间的比较。基排序是一种借助多关键字排序思想对单逻辑关键字进行排序的方法。所谓的多关键字排序就是有多个先级不同的关键字。比如说成绩的排序，如果两个人总分相同，则语文高的排在前面，语文成绩也相则数学高的排在前面。。。如果对数字进行排序，那么个位、十位、百位就是不同优先级的关键字，果要进行升序排序，那么个位、十位、百位优先级一次增加。基数排序是通过多次的收分配和收集来现的，关键字优先级低的先进行分配和收集。

**举个栗子：



**



实现代码：

```
public class RadixSort {

    public static void radixSort(int[] arr) {
        if(arr == null && arr.length == 0)
            return ;

        int maxBit = getMaxBit(arr);


        for(int i=1; i<=maxBit; i++) {

            List<List<Integer>> buf = distribute(arr, i); //分配
            collecte(arr, buf); //收集
        }

    }

    /**
     * 分配
     * @param arr 待分配数组
     * @param iBit 要分配第几位
     * @return
     */
    public static List<List<Integer>> distribute(int[] arr, int iBit) {
        List<List<Integer>> buf = new ArrayList<List<Integer>>();
        for(int j=0; j<10; j++) {
            buf.add(new LinkedList<Integer>());
        }
        for(int i=0; i<arr.length; i++) {
            buf.get(getNBit(arr[i], iBit)).add(arr[i]);
        }
        return buf;
    }
```

```
    /**
     * 收集
     * @param arr 把分配的数据收集到arr中
     * @param buf
     */
    public static void collecte(int[] arr, List<List<Integer>> buf) {
        int k = 0;
        for(List<Integer> bucket : buf) {
            for(int ele : bucket) {
                arr[k++] = ele;
            }
        }


    }

    /**
     * 获取最大位数
     * @param x
     * @return
     */
    public static int getMaxBit(int[] arr) {
        int max = Integer.MIN_VALUE;
        for(int ele : arr) {
            int len = (ele+"").length();
            if(len > max)
                max = len;
        }
        return max;
    }

    /**
     * 获取x的第n位，如果没有则为0.
     * @param x
     * @param n
     * @return
     */
    public static int getNBit(int x, int n) {

        String sx = x + "";
        if(sx.length() < n)
            return 0;
        else
            return sx.charAt(sx.length()-n) - '0';
    }

}
```

排序算法的各自的使用场景和适用场合。

排序方法	平均时间	最坏情况	辅助存储
简单排序	$O(n^2)$	$O(n^2)$	$O(1)$
快速排序	$O(n\log n)$	$O(n^2)$	$O(\log n)$
堆排序	$O(n\log n)$	$O(n\log n)$	$O(1)$
归并排序	$O(n\log n)$	$O(n\log n)$	$O(n)$
基数排序	$O(d(n+rd))$	$O(d(n+rd))$	$O(rd)$

1. **从平均时间来看，快速排序是效率最高的，但快速排序在最坏情况下的时间性能不如堆排序和归并序。而后者相比较的结果是，在n较大时归并排序使用时间较少，但使用辅助空间较多。**

2. **上面说的简单排序包括除希尔排序之外的所有冒泡排序、插入排序、简单选择排序。其中直接插入序最简单，但序列基本有序或者n较小时，直接插入排序是好的方法，因此常将它和其他的排序方法如快速排序、归并排序等结合在一起使用。**

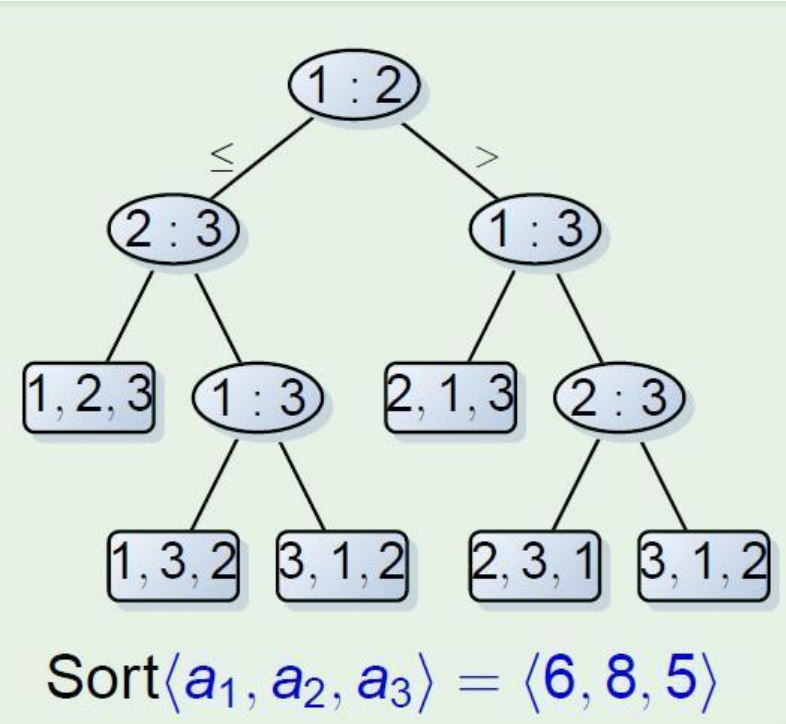
3. **基数排序的时间复杂度也可以写成O(d*n)。因此它最使用于n值很大而关键字较小的的序列。若关字也很大，而序列中大多数记录的最高关键字均不同，则亦可先按最高关键字不同，将序列分成若干的子序列，而后进行直接插入排序。**

4. **从方法的稳定性来比较，基数排序是稳定的内排方法，所有时间复杂度为O(n^2)的简单排序也是定的。但是快速排序、堆排序、希尔排序等时间性能较好的排序方法都是不稳定的。稳定性需要根据体需求选择。**

5. **上面的算法实现大多数是使用线性存储结构，像插入排序这种算法用链表实现更好，省去了移动元的时间。具体的存储结构在具体的实现版本中也是不同的。**

附：基于比较排序算法时间下限为O(nlogn)的证明：

基于比较排序下限的证明是通过决策树证明的，决策树的高度Ω（nlgn），这样就得出了比较排序的限。



**首先要引入决策树。 首先决策树是一颗二叉树，每个节点表示元素之间一组可能的排序，它予以京行的比较相一致，比较的结果是树的边。 先来说明一些二叉树的性质，令T是深度为d的二叉树，则T

多有2^片树叶。 具有L片树叶的二叉树的深度至少是logL。 所以，对n个元素排序的决策树必然有n!树叶（因为n个数有n!种不同的大小关系），所以决策树的深度至少是log(n!)，即至少需要log(n!)次较。 而 log(n!)=logn+log(n-1)+log(n-2)+...+log2+log1 >=logn+log(n-1)+log(n-2)+...+log(n/2 >=(n/2)log(n/2) >=(n/2)logn-n/2 =O(nlogn)所以只用到比较的排序算法最低时间复杂度是O(nlon)。**