1.算法的特性

有穷性：执行有穷步之后结束；

确定性：算法中每一条指令都必须有确切的含义，不能含糊不清；

输入（>=0），输出（>=1）

有效性：算法的每个步骤都能有效执行并能得到确定的结果。例如a=0，b/a就无效。

2.算法的复杂度

2.1.时间复杂度

执行所需消耗的大概时间。

O(1)

O(1)为常量的时间复杂度的统称，如不带有循环语句的程序执行所需要的时间；

二叉树深度计算：log_2n+1

一个循环体执行n次的时间复杂度：O(n)

2.2.空间复杂度

3.排序算法概念

稳定和不稳定排序：相同的值排序后不影响原来的顺序；

内排序和外排序：内排序，在内存中进行排序；外排序涉及外部存储空间。

插入类排序

3.1.直接插入排序(Straight Insertion Sort)

直接插入排序（Straight Insertion Sort）是一种最简单的排序方法，其基本操作是一个个的将记录插入到已排好的有序表中。跟冒泡排序有点类似，但区别于冒泡排序的是，直接插入排序是直到最终找到目标位置才插入当前排序的数据，故这是直接插入命名来源。

```
public class InsertSort {
    public void insertSort(int[] arr) {
        // i=0,第一个值可看作有序序列，所以从第二个值开始
        for(int i=1;i<arr.length;i++) {
            // 升序排列，判断是否有序
            if(arr[i]<arr[i-1]) {
                int temp = arr[i];//记录需要插入的数据
                int j=i;//记录需要插入的位置
                for(;j>0 && arr[j-1]>temp;j--) { // 判断已经排好序的数据与目标值比较大小，直至找到插入位置
                    arr[j]=arr[j-1];//比目标数据大的值，往后面移动
                }
                arr[j]=temp;//插入数据
```

```
            }

        }
    }

    public static void main(String[] args) {
        InsertSort insertSort = new InsertSort();
        int[] arr= {5,2,7,2,19,11,1};
        insertSort.insertSort(arr);
        for(int i=0;i<arr.length;i++) {
            System.out.print(arr[i]+",");
        }
    }

}
```

3.2.希尔排序（Shell`s Sort)

一种特殊的插入排序，希尔排序引入步长，每次比较根据步长移动，说白了直接插入排序全程都是以步长为1的一个“希尔排序”。

以下是以数组长度减半的步长的希尔排序算法：

当步长gap=1时，希尔排序就变成了直接插入排序

```
import java.util.Arrays;

public class ShellSort {

    /**
     * 希尔排序
     * @param arr
     */
    public static void shellSort(int[] arr) {
        //遍历所有的步长，每次减半
        for(int gap = arr.length>>1;gap>0;gap>>=1){
            shellInsert(arr, gap);
        }

    }
    /**
     * 带有步长的插入排序，当gap=1时，其实就是普通的插入排序
     * @param arr 排序数组
     * @param gap 步长
     */
    public static void shellInsert(int[] arr,int gap) {
        //插入排序的处理逻辑
        for (int i = gap; i < arr.length; i++) {
            //升序排序，如果前一个步长的值大于当前的值，则需要往前移动，直至左<=右
            if(arr[i]<arr[i-gap]) {
                int temp = arr[i];//记录需要插入的数据
                int j=i;//初始化需要插入的位置，用于记录插入位置
```

```
                while(j >=gap && arr[j-gap]>temp) {
                    arr[j] = arr[j-gap];//比目标数据大的值，交换位置往后面移动
                    j -= gap;//一直往前移动
                }
                arr[j]=temp;//插入数据
            }
        }
        // 每次排序后遍历
        System.out.println(Arrays.toString(arr));
    }

    public static void main(String[] args) {
         int[] arr = { 8, 1, 2, 7, 3, 5, 0, 4, 6};
         // 希尔排序（优化后的插入排序）
         shellSort(arr);
         // 插入排序：gap=1
//         shellInsert(arr,1);
    }

}
```

交换类排序

3.3.冒泡排序（Bubble Sort)

冒泡排序每次对比只要复合条件都会执行存储位置交换操作，从第一个对比到最后一个，全部元素此操作直至最后一个元素（最后一个不用对比）。

3.4.快速排序（Quick Sort)

采用分治法，基本思想是将原问题分解成若干个规模更小但结构与原问题相似的自问题。通过递归地解决这些子问题，然后再将这些子问题的解结合合成原问题的解。

快速排序的操作步骤：

1. 在待排序的n个记录中任取一个记录，以该记录的排序码为准，将所有记录都分成两组，第1组都小于该数，第2组都大于该数；
2. 采用相同的方法对左、右两组分别进行排序，直到所有记录都排到相应位置为止。

```
import java.util.Arrays;

public class QuickSort {

    public static void quickSort(int[] arr,int left,int right) throws Exception {
        if(arr==null || arr.length<1)
            throw new Exception("排序的数组不能为空");
        if(right-left==0)
            return;
        //取数组第一个值
        int pivot = arr[left];
        //保存原始下标值
```

```
        int start=left;
        int end=right;
        while(left<right) {
            // 找右侧比pivot小的数据，没找到就一直自减，直至找到或下标出现left>=right
            while(left<right && arr[right]>=pivot)
                right--;
            if(left<right)
                arr[left]=arr[right];
            else
                break;
            // 找左侧比pivot大的数据，没找到就一直自增，直至找到或下标出现left>=right
            while(left<right && arr[left]<=pivot)
                left++;
            if(left<right)
                arr[right]=arr[left];
            else
                break;
        }
        arr[right]=pivot;
        System.out.println(Arrays.toString(arr));

        if(left-1>start)
            quickSort(arr, start, right-1);
        if(right+1<end)
            quickSort(arr, right+1,end);
    }

    public static void main(String[] args) throws Exception {
        int[] arr = { 15, 13, 16, 19, 18, 3, 12, 17};
        System.out.println("未排序前："+Arrays.toString(arr));
        quickSort(arr, 0, arr.length-1);
    }

}
```

每次以某个元素作为基准，以该基准对数组进行二分；

其中第一次的基准值为树的根节点，二分后形成左右子树，左子树都小于根节点，右子树都大于根节点，又名为排序二叉树

一直递归执行，直至叶子节点（无子结点），最终形成的是普通的排序二叉树的分布。

其中排序二叉树中序遍历为递增数列，即是快速排序的最终结果。

二叉树与快速排序算法的关系：https://www.jianshu.com/p/e55dabe60cf5

选择类排序

3.5.简单选择排序（Simple Selection Sort）

选择未排序的序列中**最小的值或最大的值**跟未排序的第一个值做交换，直至最后一个元素。

区别于冒泡排序，直接选择排序在寻找最值时不做交换存储位置，只是找到最值后再跟第一个值做交换存储位置，而冒泡排序每次对比只要复合条件都会执行存储位置交换操作。

```
import java.util.Arrays;

public class SelectSort {

    public static void main(String[] args) {
       int[]arr={66,35,22,11,5};
       selectSort(arr);
   }

   // 选择排序
   public static  void selectSort(int[] arr) {
       if (arr == null)
           return;
       // 最后一个无需再排序：arr.length - 1
       for (int i = 0; i < arr.length - 1; i++) {
           int minValue = arr[i]; //记录找到的最小值
           int tempIndex = i;    //记录找到的最小值的下标
           // 一个个比较，寻找最小值
           for (int j = i + 1; j < arr.length; j++) {
               if (minValue > arr[j]) {
                   minValue = arr[j];
                   tempIndex = j;
               }
           }
           // 交换值：将找到的最小值放在下标i中
           if (tempIndex!=i) {
               arr[tempIndex] = arr[i];
               arr[i] = minValue;
           }
           System.out.println("第"+ (i+1) +"轮后：" + Arrays.toString(arr));
       }
   }
}
```

3.6.堆排序（Heap Sort）

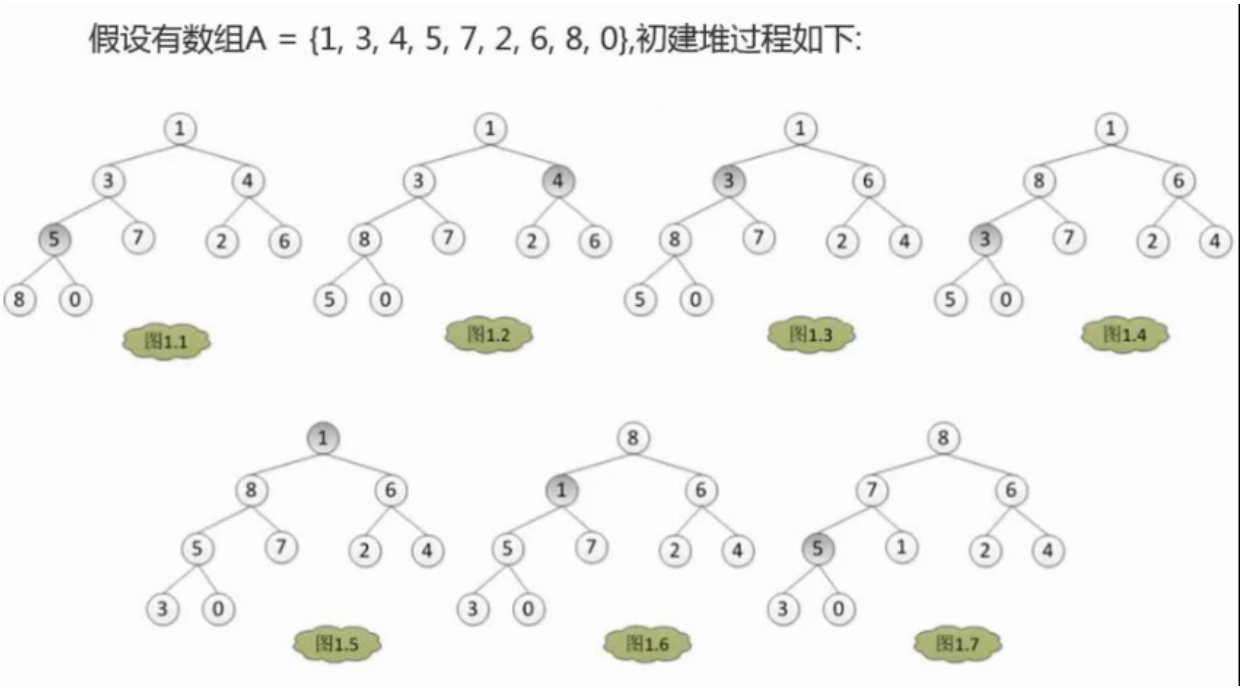
概念：设有n个元素的序列{K_1,K_2,...,K_n}，当且仅当满足下述关系之一时，称为堆。

小顶堆：k_i≤k{2i} 且 k_i≤k{2i+1}，其根节点存储最小值，从小到大排序；

大顶堆：k_i≥k{2i} 且 k_i≥k{2i+1}，其根节点存储最大值，从大到小排序；

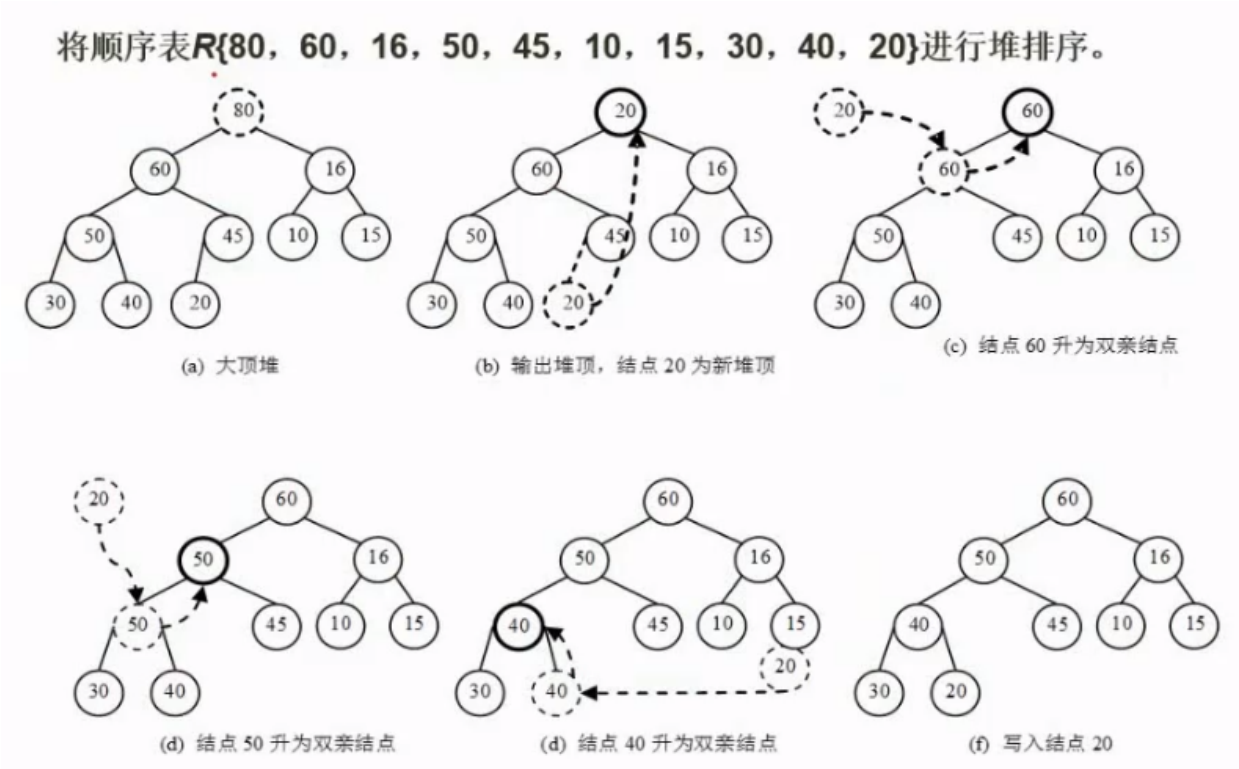
初建堆

例如，大顶堆的建堆过程



取最值、重建堆

将最大值取出来（剪掉根节点），将最后的一个叶子结点填充到原根节点，再进行重建堆操作。



代码实现

```
import java.util.Arrays;

/**
 * 堆排序
 */
public class HeapSort {
```

```
    public void heapSort(int arr[]) {
        int n = arr.length;

        // 初建堆：对数组进行初次建堆（排序）
        for (int i = n / 2 - 1; i >= 0; i--)
            heapify(arr, n, i);

        // 一个一个地从堆中提取最值元素：根结点为最值，剪根结点
        for (int i = n - 1; i >= 0; i--) {
            // 移动当前根结点到当前数组长度的末尾
            int temp = arr[0];
            arr[0] = arr[i];
            arr[i] = temp;

            // 堆元素减少一个(i=n-1)，再重建堆（跟初建堆操作一样）
            heapify(arr, i, 0);
        }
    }

    // 将结点i作为根的子树堆起来，n 是堆的大小
    public void heapify(int arr[], int n, int i) {
        int largest = i; // 将最大值初始化为根
        // 完全二叉树的特性：跟结点为i时，左子结点下标为：left = 2*i；右子结点下标为：right =
2*i + 1
        // 因为数组下标从0开始，所以下标要多加1，即left = 2 * i + 1，right = 2 * i + 2
        int left = 2 * i + 1;
        int right = 2 * i + 2;

        // 先判断是否有左子树：left < n，再判断左子结点大于根结点
        if (left < n && arr[left] > arr[largest])
            largest = left;

        // 先判断是否有右子树：right < n，再判断右子结点大于根结点
        if (right < n && arr[right] > arr[largest])
            largest = right;

        // 如果最大不是根，则进行交换
        if (largest != i) {
            int swap = arr[i];
            arr[i] = arr[largest];
            arr[largest] = swap;

            // 递归地对子树进行堆化
            heapify(arr, n, largest);
        }
    }

    // 测试
    public static void main(String args[]) {
        int arr[] = {15, 10, 13, 20, 6, 7};
```

```
        HeapSort heapSort = new HeapSort();
        heapSort.heapSort(arr);

        System.out.println("堆排序结果：" + Arrays.toString(arr));
    }
}
```

应用于取值极值，比如获取排名中前三名。

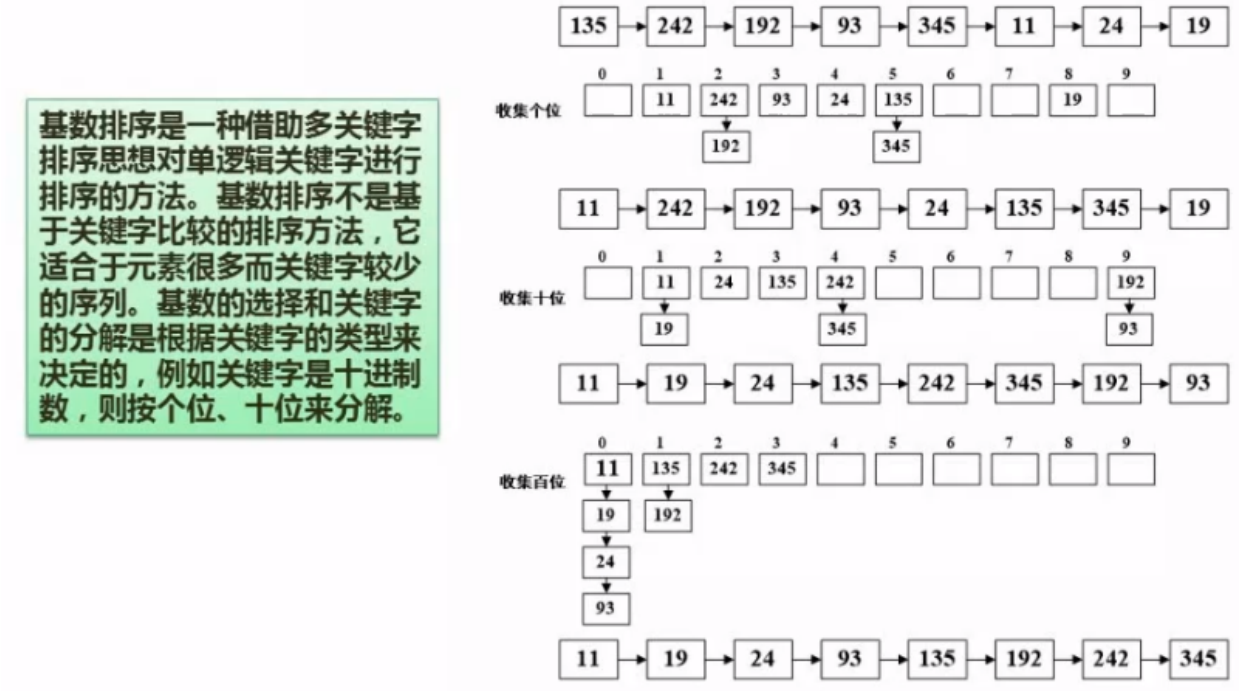
3.7.归并排序（Merge Sort）

将序列拆分成两个两个进行排序，拆到只有一个元素，再两两有序序列合并成新的有序序列，直至所有序列被合并，最终得到的就是有序的序列，归并排序又称为合并排序。

```
import java.util.Arrays;

public class MergeSort {

    public static void main(String[] args) {
        int[] arr = {12, 20, 5, 16, 15, 1, 30, 45, 23, 9};
        System.out.println("排序前:" + Arrays.toString(arr));
        mergeSort(arr, 0, arr.length - 1);
        System.out.println("排序后:" + Arrays.toString(arr));
    }

    // 归并排序
    public static void mergeSort(int[] arr, int left, int right) {
        int middle = (left + right) / 2;
        // 递归结束条件为left>=right，所有子序列的长度都为1，即数组被拆分为只有一个一个元素了，然后再归并
        if (left < right) {
            // 处理左边
            mergeSort(arr, left, middle);
            // 处理右边
            mergeSort(arr, middle + 1, right);
            // 归并
            merge(arr, left, middle, right);
        }
    }

    /**
     *  将两个有序序列合并成一个有序序列
     *  当序列只有一个元素时，可以看作是一个有序序列，这也是合并的开始
     * @param arr 原数组
     * @param left 左指针
     * @param middle 中间指针
     * @param right 右指针
     */
    public static void merge(int[] arr, int left, int middle, int right) {
        System.out.println("left="+left+",middle="+middle+",right="+right);
```

```
        int[] temp = new int[right - left + 1];
        int i = left; // 左边数组下标
        int j = middle + 1; // 右边数组下标
        int k = 0;
        /*
         * 把较小的数移到新数组中
         * 哪边数据小的，下标就要往右移动（即自增），并记录该数据，直至其中一边没有元素比较，则得出两个有序序列的合并后的顺序
         * 例子：i组 [5, 12],j组[15, 16]：比较5<15，i就要自增，再比较12<15，i再自增（结果i>middle了），则得出的顺序为：5，12，15
         *          因为合并的序列本身是有序序列，所以剩下没比较的是大于前面的数据的，后边的操作只要移入数组即可，最终得到合并后的有序序列为：5，12，15，16
         */
        while (i <= middle && j <= right) {
            temp[k++] = arr[i] < arr[j] ? arr[i++] : arr[j++];
        }
        // 后续的两个while循环，每次只会有一个会被执行
        // 把左边这组有序序列剩余的元素移入数组
        while (i <= middle) {
            temp[k++] = arr[i++];
        }
        // 把右边这组有序序列剩余的元素移入数组
        while (j <= right) {
            temp[k++] = arr[j++];
        }
        // 把新数组中的数据覆盖原数组对应下表的数据
        for (int k2 = 0; k2 < temp.length; k2++) {
            arr[k2 + left] = temp[k2];
        }
        // 每次合并的结果
        System.out.println(Arrays.toString(arr));
    }

}
```

3.8.基数排序（Radix Sort）



4.查找算法

4.1.顺序查找算法

从头到尾一个个进行匹配。

```
    // 顺序查找
    public int demo1(int[] arr,int val) {
        for(int i=0;i<arr.length;i++) {
            if(arr[i]==val)
                return val;
        }
        return -1;
    }
```

4.2.二分查找算法（折半查找）

参考文章：二分查找又叫折半查找，是一种简单又快速的查找算法

时间复杂度O(log_2n)

```
public class Sort {

    public static void main(String[] args) throws Exception {

        Sort sort = new Sort();
        int[] arr= {1,5,2,7,2,19,11};
        int demo1 = sort.demo1(arr, 19);
        System.out.println(demo1);
        System.out.println("排序前: ");
        sort.printfDemo(arr);
```

```
        //sort.sortArr(arr);
        //System.out.println("排序后：");
        //sort.printfDemo(arr);
        int demo2 = sort.demo2(arr, 19);
        System.out.println("排序后：");
        sort.printfDemo(arr);
        System.out.println("二分查找结果下标："+demo2+",下标对应的值："+arr[demo2]);
    }

    // 顺序查找
    public int demo1(int[] arr,int val) {
        for(int i=0;i<arr.length;i++) {
            if(arr[i]==val)
                return val;
        }
        return -1;
    }

    // 二分查找：需要先进行排序，然后再进行查找；前提是该序列支持排序，否则无法使用
    public int demo2(int[] arr,int val) throws Exception {
        if(arr==null)
            throw new Exception("查找的数据不能为NULL!");
        //特殊情况
        if(arr.length==1 && arr[0]==val) {
            return arr[0];
        }
        // 当数组大小没有达到max 大小时，可以考虑使用简单的查找方法，这样更高效
        int max  = 5;
        if(arr.length<max) {
            return demo1(arr,val);
        }
        // 正式执行二分查找算法
        // 1.先排序
        sortArr(arr);
        // 2.再查找：查询到目标数据的下标值
//        int targetIndex = binarySearch(arr,0,arr.length, val);
        int targetIndex = binarySearch(arr, val);
        return targetIndex;
    }

    // 递归方法实现二分查找，不适用于数据量太大的情况
    public int binarySearch(int[] arr,int left,int right, int target) {
        if(left>right)
            return -1;
        else {
            // 取中间值，当arr长度为奇数时，向下取整
            int  mid= (left+right)>>1;
            int midNum = arr[mid];
            // 在二分的左边
            if(target<midNum) {
                return binarySearch(arr,left,mid-1, target);
            }
```

```
            // 在二分的右边
            else if(target>midNum) {
                return binarySearch(arr, mid+1, right, target);
            }
            else {
                return mid;
            }
        }

    }

    public int binarySearch(int[] arr,int target) {
        int left=0;
        int right=arr.length;
        while(left<=right) {
            // 取中间值，当arr长度为奇数时，向下取整
            int  mid= (left+right)>>1;
            int midNum = arr[mid];
            // 在二分的左边
            if(target<midNum) {
                right = mid-1;
            }
            // 在二分的右边
            else if(target>midNum) {
                left = mid+1;
            }
            else {
                return mid;
            }
        }
        return -1;
    }

    public void sortArr(int[] arr) {
        int tmp = 0;
        for(int i=0;i<arr.length;i++) {
            for(int j=i+1;j<arr.length;j++) {
                if(arr[i]>arr[j]) {
                    tmp = arr[i];
                    arr[i]=arr[j];
                    arr[j]=tmp;
                }
            }
        }
    }

    //遍历
    public void printfDemo(int[] arr) {
        for(int i=0;i<arr.length;i++) {
            System.out.print(arr[i]+",");
        }
        System.out.println();
```

```
    }
}
```

4.3.查找散列表

按照一定的规则进行存储，例如对需要存储的数据n取模n%3=?，然后将数据保存到指定位置。在查找时，再对数据进行取模运算n%3 就可以得到具体位置了。其中，对模取3，就是可以解决散列冲突，但只能解决2次散列冲突。

散列冲突解决办法有：线性探测法和伪随机数法。比如再次进行散列操作。

5.有趣的算法题

5.1.汉诺塔

汉诺塔移动次数：2^(n-1) + 2^(n-2) + ... +2^1+2^0=2^n - 1

完成哪一层，二进制对应的位数就是哪一个 例如：3层汉诺塔 对应二进制就是111，100就是第三层移动完成，110就是第二层移动完成，111就是三层都移动完成； 结论：汉诺塔的层号对应二进制的位数

汉诺塔延申： 汉诺塔明显和二进制存在必然关联 关联一、本质都是只有两种状态：移动和没移动对应二进制1和0； 关联二、当第n层被移动时，上面的第(n-1)层至第1层都要置0，这是二进制发生进位时，后面的都要归零一样

```
public class Hanoi {
    // 阶数
    private static int n = 4;
    //验证汉诺塔移动次数
    private static int sum=0;
    public static void main(String[] args) {
        System.out.println(String.format("%s层汉诺塔的移动顺序：", n));
        move(n, 'A','B','C');
        System.out.println("汉诺塔移动次数："+sum);
        sum = moveTree(n, 'A','B','C');
        System.out.println("汉诺塔移动次数："+sum);
    }

    /**
     * (n-1) A -> B
     *   n   A -> C
     * (n-1) B -> C
     *
     * 结束条件为：当n=1 时，  A -> C
     */
    public static void move(int n,char A, char B, char C) {
        if(n==1) {
            System.out.println(A + " -> " + C);
            sum++;
        }
        else {
            move(n-1, A, C, B);//每次都是输出A->C，所以要看到A->B，就需要将B和C交换
```

```
            if(n==Hanoi.n)
                System.out.println("前面完成(n-1)层：从A移动到B");
            System.out.println(A + " -> " + C);
            if(n==Hanoi.n)
                System.out.println("完成第(n)层：从A移动到C");
            sum++;

            move(n-1, B, A, C);//每次都是输出A->C，所以要看到B->C，就需要将A和B交换
            if(n==Hanoi.n)
                System.out.println("前面完成(n-1)层：从B移动到C");
        }
    }

    /**
     * 汉诺塔与满二叉树
     * (n-1) A -> B
     *   n   A -> C
     * (n-1) B -> C
     *
     * 结束条件为：当n=1 时，  A -> C
     */
    public static int moveTree(int n,char A, char B, char C) {
        if(n==1)
            System.out.println(String.format("第 %s 层(叶子节点)：%s -> %s",n, A, C));
        else {
            moveTree(n-1, A, C, B);//每次都是输出A->C，所以要看到A->B，就需要将B和C交换

            if(n==Hanoi.n)
                System.out.println(String.format("第 %s 层(根节点)：%s -> %s", n, A,
C));
            else
                System.out.println(String.format("第 %s 层(分支结点)：%s -> %s", n, A,
C));

            moveTree(n-1, B, A, C);//每次都是输出A->C，所以要看到B->C，就需要将A和B交换
        }
        //汉诺塔的移动次数为：2^n-1
        return (int) Math.pow(2, n)-1;
    }

}
```

汉诺塔递归式：

```
T(n)=
\begin{cases}
2T(n-1)+1\quad n≥0 \\
\quad 1 \quad n=1
\end{cases}
```

由递归式求n阶汉诺塔移动次数：

由递归式可知：

```
T(n)=2T(n-1)+1\\
\quad\quad\quad\quad\quad\quad=2(2T(n-2)+1)+1\\
\quad\quad\quad\quad\quad=2^2T(n-2)+2+1\\
\quad\quad\quad\quad\quad\quad\quad=2^3T(n-3)+2^2+2+1\\
\quad\quad\quad\quad\quad\quad\quad\quad=2^{n-1}T(1)+2^{n-2}+...+2+1
```

又因当n=1时，T(1)=1，得：T(n)=2^{n-1}+2^{n-2}+...+2^1+2^0=2^n-1

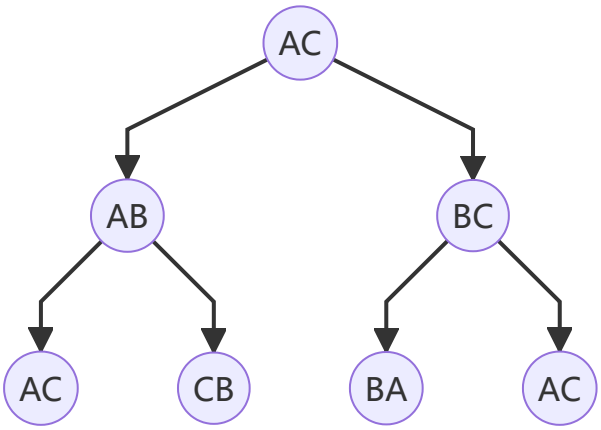
解得n阶汉诺塔移动次数为：2^n-1 次。

结合二进制可知，**n阶汉诺塔移动次数等于n位二进制得最大值**，如：4阶汉诺塔移动次数为1111*{2}=15*{10}

当有了这一层结论时，求汉诺塔m阶第一次被移动时，需要移动的次数为2^{m-1}，如：4阶汉诺塔的第3阶第一次被移动时，需要移动的次数为4。

汉诺塔与满二叉树

3阶汉诺塔对应的满二叉树：



3阶汉诺塔的移动步骤为满二叉树的**中序遍历**：AC、AB、CB、AC、BA、BC、AC

通过代码验证

```
public class Hanoi {
    // 阶数
    private static int n = 3;
    public static void main(String[] args) {
        System.out.println(String.format("%s层汉诺塔的移动顺序：", n));
        int sum = moveTree(n, 'A','B','C');
        System.out.println("汉诺塔移动次数："+sum);
    }


    /**
     * 汉诺塔与满二叉树
     * (n-1) A -> B
```

```
     *   n   A -> C
     * (n-1) B -> C
     *
     * 结束条件为：当n=1 时，  A -> C
     */
    public static int moveTree(int n,char A, char B, char C) {
        if(n==1)
            System.out.println(String.format("第 %s 层(叶子节点)：%s -> %s",n, A, C));
        else {
            moveTree(n-1, A, C, B);//每次都是输出A->C，所以要看到A->B，就需要将B和C交换

            if(n==Hanoi.n)
                System.out.println(String.format("第 %s 层(根节点)：%s -> %s", n, A,
C));
            else
                System.out.println(String.format("第 %s 层(分支结点)：%s -> %s", n, A,
C));

            moveTree(n-1, B, A, C);//每次都是输出A->C，所以要看到B->C，就需要将A和B交换
        }
        //汉诺塔的移动次数为：2^n-1
        return (int) Math.pow(2, n)-1;
    }

}
```

> 3层汉诺塔的移动顺序：
>
> 第 1 层(叶子节点)：A -> C
>
> 第 2 层(分支结点)：A -> B
>
> 第 1 层(叶子节点)：C -> B
>
> 第 3 层(根节点)：A -> C
>
> 第 1 层(叶子节点)：B -> A
>
> 第 2 层(分支结点)：B -> C
>
> 第 1 层(叶子节点)：A -> C
>
> 汉诺塔移动次数：7

从输出结果可以看到，汉诺塔**盘子编号**对应满二叉树**自底向上**计算的**层号**，如：盘子1对应叶子节点。

为了更好理解，可以写成这样：

```
    public static int moveTree(int n,char A, char B, char C) {
        if(n==1)
            System.out.println(String.format("第 %s 层(叶子节点)：%s -> %s",Hanoi.n-
n+1, A, C));
        else {
            moveTree(n-1, A, C, B);//每次都是输出A->C，所以要看到A->B，就需要将B和C交换
```

```
            if(n==Hanoi.n)
                System.out.println(String.format("第 %s 层(根节点)：%s -> %s",
Hanoi.n-n+1, A, C));
            else
                System.out.println(String.format("第 %s 层(根节点)：%s -> %s",
Hanoi.n-n+1, A, C));

            moveTree(n-1, B, A, C);//每次都是输出A->C，所以要看到B->C，就需要将A和B交换
        }
        //汉诺塔的移动次数为: 2^n-1
        return (int) Math.pow(2, n)-1;
    }
```

汉诺塔递归实现与二叉树中序遍历的递归实现，在代码实现上很类似

```
public static void inorder(TreeNode root) {
    if (root == null)
        return;
    inorder(root.left);
    System.out.print(root.val);
    inorder(root.right);
}
```

汉诺塔的移动步骤可以用满二叉树的中序遍历来表示，反过来，我们可以通过满二叉树的特性推导出汉诺塔的一些特性：

- 满二叉树总的结点数为2^n-1，所以汉诺塔移动次数为2^n-1；
- 满二叉树第n层的节点数为2^{n-1}，所以n阶汉诺塔第i个盘子被移动的次数为2^{(n-i+1)-1}=2^{n-i}；
- 满二叉树叶子节点数为2^{n-1},所以汉诺塔第一个盘子被移动的次数为2^{n-1}；
- 满二叉树是二进制的一种表现形式，所以汉诺塔也是二进制的一种表现形式，其中汉诺塔的移动过程就是二进制的累加过程。

还有很多特性，不一一列出，其推导过程的意义不仅仅是得出汉诺塔有哪些特性，更重要的是明白递归