一、插入类排序

1.1 直接插入

```
ic static void sort(int[] nums){
for (int i = 1; i < nums.length; i++) {
    if (nums[i] < nums[i - 1]){
        int j = i - 1;
        int temp = nums[i];
        while ( j >= 0 && nums[j] > temp){
            nums[j + 1] = nums[j];
            j--;
        }
        nums[j + 1] = temp;
    }
}

```

时间复杂度：最好O(n)，最坏O(n^2)，平均O(n^2)

空间复杂度： O(1)

1.2 折半插入

```
ic static void sort(int[] nums){
for (int i = 1; i < nums.length; i++) {
    int temp = nums[i];
    if (temp < nums[i - 1]){
        int low = 0;
        int high = i - 1;
        while (low <= high){
            int mid = (low + high) / 2;
            if (nums[mid] > temp){
                high = mid - 1;
            }else {
                low = mid + 1;
            }
        }
        for (int j = i - 1; j >= low ; j--) {
            nums[j + 1] = nums[j];
        }
        nums[low] = temp;///折半查找low是查找失败元素所处的位置
    }
}

```

时间复杂度：O(n^2)

空间复杂度：O(1)

插入排序是稳定的排序，因为每回都是从后往前比较，然后移动，所以不会出现相同元素相对位置发生变化。

1.3 希尔排序

```
ic static void sort(int[] nums){
    int length = nums.length;
    int d;
    for (d = length/2; d > 0; d/=2) {
        for (int i = d; i < length; i++) {
            if (nums[i] < nums[i - d]) {
                int temp = nums[i];
                int index = i - d;
                while (index >= 0 && nums[index] > temp){
                    nums[index + d] = nums[index];
                    index-= d;
                }
                nums[index + d] = temp;
            }
        }
    }
}
```

空间复杂度：O(1)

时间复杂度：依赖于增量序列的函数，当n在某个特定范围内，时间复杂度为O(`n^1.3`)，最坏情况下为O(n^2)

稳定性：当相同关键字的记录被划分到不同的子表时，可能会改变它们之间的相对次序，因此希尔排序是一个不稳定的排序方法。

二、交换类排序

2.1 冒泡排序

```
ic static void sort(int[] nums){
int length = nums.length;
boolean tag;
for (int i = 1; i < length; i++) {
    tag = false;
    for (int j = 0; j < length - i; j++) {
        if (nums[j] > nums[j + 1]){
            int temp = nums[j];
            nums[j] = nums[j + 1];
            nums[j + 1] = temp;
            tag = true;
        }
    }
    if (!tag){
        break;
    }
```

```
}

```

时间复杂度：最好O(n)，最坏O(n^2)，平均O(n^2)

空间复杂度：O(1)

稳定的排序：相等的时候不会交换元素位置

2.2 快速排序

```
ic static void sort(int[] nums,int low, int high){
int i = low;
int j = high;
if (low < high){
    int temp = nums[low];
    while (i != j){
        while (i < j && nums[j] >= temp){
            j --;
        }
        if (i < j){
            nums[i] = nums[j];
            i++;
        }
        while (i < j && nums[i] <= temp){
            i ++;
        }
        if (i < j){
            nums[j] = nums[i];
            j--;
        }
    }
    nums[i] = temp;
    sort(nums, low, i - 1);
    sort(nums, i + 1, high);
}

```

时间复杂度：O(nlog2n)，最坏O(n^2)

空间复杂度：O(log2n)，最坏O(n)

不稳定，会移动相等元素

2.3 三向切分的快速排序

```
ate static void sortBy3Way(int[] array, int low, int high) {


int lt = low;
int gt = high;
int i = low + 1;
```

```
if (low < high) {
    //切分的元素
    int temp = array[low];
    while (i <= gt) {
        if (array[i] < temp) {
            swap(array, lt++, i++);
        } else if (array[i] > temp) {
            swap(array, i, gt--);
        } else {
            i++;
        }
    }
    //递归
    sortBy3Way(array, low, lt - 1);
    sortBy3Way(array, gt + 1, high);
}


ate static void swap(int[] array, int i, int i1) {
int temp = array[i];
array[i] = array[i1];
array[i1] = temp;

```

对于**包含大量重复元素的数组**，三向切分的快速排序算法将排序时间从线性对数级降低到线性级别，因此时间复杂度介于O(N)和O(Nlg N)之间，这依赖于输入数组中重复元素的数量。

2.4 一次遍历的快速排序

```
age com.sort.exchange;


Author: 98050
Time: 2019-05-21 14:07
Feature:

ic class QuickSort2 {

public static void main(String[] args) {
    int[] array = new int[]{8,1,3,5,2,5,8,10};
    sort(array,0,array.length - 1);
    for (int i : array){
        System.out.println(i);
    }
}

public static void sort(int[] nums,int low, int high){
    if (low < high) {
        int i = -1;
        int temp = nums[high];
        for (int j = 0; j < high; j++) {
            if (nums[j] < temp) {
```

```
                swap(nums, ++i, j);
            }
        }
        swap(nums, ++i, high);
        sort(nums, low, i - 1);
        sort(nums, i + 1, high);
    }
}

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

```

三、选择类排序

3.1 简单选择排序

```
ate static void sort(int[] array) {
for (int i = 0; i < array.length; i++) {
    int k = i;
    for (int j = i + 1; j < array.length; j++) {
        if (array[k] > array[j]){
            k = j;
        }
    }
    int temp = array[i];
    array[i] = array[k];
    array[k] = temp;
}

```

时间复杂度：O(n^2)

空间复杂度：O(1)

稳定性：在第i趟找到最小元素后，和第i个元素交换，可能导致第i个元素与其含有相同关键字元素的相对位置发生改，所以是一个不稳定的排序。

3.2 堆排序

```
public static void heapSort(int[] nums,int n){
    //初始建堆
    for (int i = n / 2; i >= 0 ; i--) {
        shift(nums,i,n);
    }
    for (int i = n; i >= 0 ; i--) {
        int temp = nums[0];
```

```java
        nums[0] = nums[i];
        nums[i] = temp;
        shift(nums,0,i - 1);
    }
}

private static void shift(int[] nums, int low, int high) {
        int root = low;
        int left = root * 2 + 1;
        while (left <= high){
            if (left < high && nums[left] > nums[left + 1]){
                left++;
            }
            if (nums[root] > nums[left]){
                int temp = nums[root];
                nums[root] = nums[left];
                nums[left] = temp;
            }
            root = left;
            left = root * 2 + 1;
        }
}
```

时间复杂度：O(nlog2n)

空间复杂度：O(1)

建堆时间：O(n)

如果仅从代码上直观观察，会得出构造二叉堆的时间复杂度为O(nlogn)的结果，这个结果是错的，虽然该算法外层套一个n次循环，而内层套一个分治策略下的logn复杂度的循环，该思考方法犯了一个原则性错误，那就是构建二叉堆是**自下而上的构建**，每一层的最大纵深总是小于等于树的深度的，因此，该问题是叠加问题，而非递归问题。那么换个方式，假如我们自上而下建立二叉堆，那么插入每个节点都和树的深度有关，并且都是不断的把 树折半来实现插入，因此是典型的递归，而非叠加。

在做证明之前，我们的前提是，建立堆的顺序是bottom-top的。 正确的证明方法应当如下：

具有n个元素的平衡二叉树，树高为logn，我们设这个变量为h。 最下层非叶节点的元素，只需做一次线性运算便可以确定大根，而这一层具有$2^{(h-1)}$个元素，我们假定O(1)=1，那么这一层元素所需时间为$2^{(h-1)} \times 1$。 由于是bottom-top建立堆，因此在调整上层元素的时候，并不需要同下层所有元素做比较，只需要同其中之一分支作比较，而作比较次数则是树的高度减去当前节点的高度。因此，第x层元素的计算量为$2^{(x)} \times (h-x)$。 又以上通项公式可得知，构造树高为h的二叉堆的精确时间复杂度为： $S = 2^{(h-1)} \times 1 + 2^{(h-2)} \times 2 + \ldots\ldots +1 \times (h-1)$ ① 通过观察第四步得出的公式可知，该求和公式为等差数列和等比数列的乘积，因此用错位想减发求解，给公式左右两侧同时乘以2，可知： $2S = 2^h \times 1 + 2^{(h-1)} \times 2+ \ldots\ldots +2 \times (h-1)$ ②

用②减去①可知： $S =2^h \times 1 - h +1$ ③

将$h = \log n$ 带入③，得出如下结论：

$S = n - \log n +1 = O(n)$

结论：构造二叉堆的时间复杂度为线性得证。

从上到下建堆，时间复杂度为O(nlog2n)

选择排序都是不稳定，交换的时候可能发生相对位置的改变！

四、归并排序

```
private static void sort(int[] array, int start, int end,int[] temp) {
    if (start >= end){
        return;
    }
    int mid = (start + end) / 2;
    sort(array, start, mid,temp);
    sort(array, mid + 1, end,temp);
    merge(array,start,mid,end,temp);
}
private static void merge(int[] array, int start, int mid, int end,int[] temp) {
    //1.第一个有序区间的开始索引
    int i = start;
    //2.第二个有序区间的开始索引
    int j = mid + 1;
    //3.临时区域的索引
    int k = start;
    while (i <= mid && j <= end){
        if (array[i] <= array[j]){
            temp[k ++] = array[i ++];
        }else {
            temp[k ++] = array[j ++];
        }
    }

    while (i <= mid){
        temp[k ++] = array[i ++];
    }
    while (j <= end){
        temp[k ++] = array[j ++];
    }

    //将排序好的元素放入元数组
    for (int l = start; l <= end ; l++) {
        array[l] = temp[l];
    }
}
```

时间复杂度：O(nlog2n)

空间复杂度：O(n)

稳定性：在merge的时候是不会改变相同关键字的相对次序的，所以是稳定的。

五、基数排序