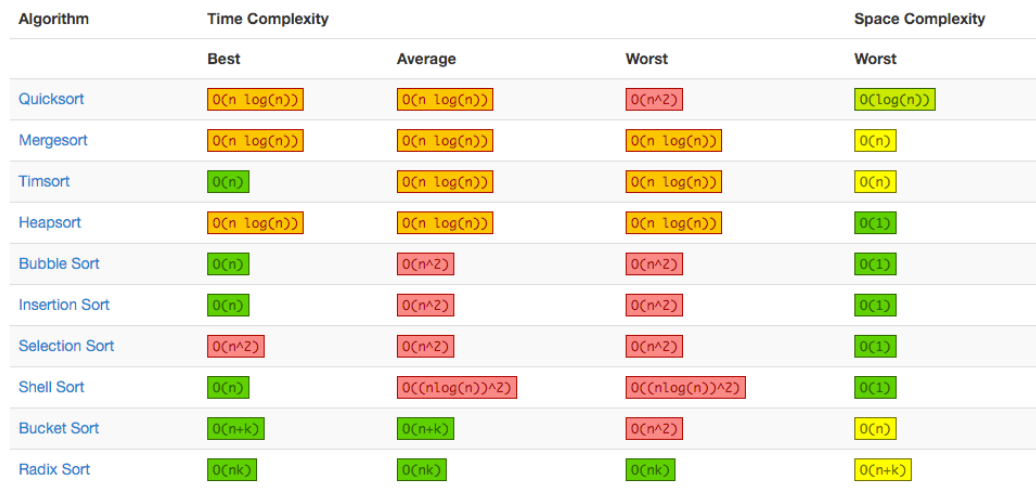
# 排序算法



直接**插入排序**

希尔排序

简单选择排序

**堆排序**

冒泡排序

**快速排序**

**归并排序**

**计数排序**

**基数排序**

**桶排序**

## 1.1 基于比较排序

### 1.1.1 直接插入排序

“**往有序的数组中快速插入一个新的元素**”

算法思想：把要排序的数组分为了两个部分, 一部分是数组的全部元素(除去待插入的元素), 另一部分是待插入的元素; 先将第一部分排序完成, 然后再插入这个元素. 其中第一部分的排序也是通过再次拆分为两部分来进行的。

类型可分为：直接插入排序、这般插入排序(或二分插入排序)、链表插入排序、希尔排序。

1. 基本思想

直接插入排序的基本思想是：将数组中的所有元素依次跟前面已经排好的元素相比较，如果选择的元素比已排序的元素小，则交换，直到全部元素都比较过为止。

1. 算法描述

插入排序一般采用in-place在数组上实现。

具体算法描述：

①. 从第一个元素开始，该元素可以认为已经被排序

②. 取出下一个元素，在已经排序的元素序列中从后向前扫描

③. 如果该元素（已排序）大于新元素，将该元素移到下一位置

④. 重复步骤3，直到找到已排序的元素小于或者等于新元素的位置

⑤. 将新元素插入到该位置后

⑥. 重复步骤②~⑤

Note: 如果比较操作的代价比交换操作大的话，可以采用二分查找法来减少比较操作的数目。该算法可以认为是插入排序的一个变种，称为二分查找插入排序。

1. 代码实现
2. /\*\*
3. \* 插入排序
4. \*
5. \* 1. 从第一个元素开始，该元素可以认为已经被排序
6. \* 2. 取出下一个元素，在已经排序的元素序列中从后向前扫描
7. \* 3. 如果该元素（已排序）大于新元素，将该元素移到下一位置
8. \* 4. 重复步骤3，直到找到已排序的元素小于或者等于新元素的位置
9. \* 5. 将新元素插入到该位置后
10. \* 6. 重复步骤2~5
11. \* @param arr 待排序数组
12. \*/
13. public static void insertionSort(int[] arr){
14. for( int i=0; i<arr.length-1; i++ ) {
15. for( int j=i+1; j>0; j-- ) {
16. if( arr[j-1] <= arr[j] )
17. break;
18. int temp = arr[j]; //交换操作
19. arr[j] = arr[j-1];
20. arr[j-1] = temp;
21. System.out.println("Sorting: " + Arrays.toString(arr));
22. }
23. }
24. }

直接插入排序复杂度：

| **平均时间复杂度** | **最好情况** | **最坏情况** | **空间复杂度** |
| --- | --- | --- | --- |
| O(n²) | O(n²) | O(n²) | O(1) |

Note: 由于直接插入排序每次只移动一个元素的位， 并不会改变值相同的元素之间的排序， 因此它是一种稳定排序。

### 1.1.2 堆排序(Heap Sort)

堆的定义如下：n个元素的序列{k1,k2,···,kn}，当且仅当满足下关系时，称之为堆。

ki <= k(2i) 且 ki <= k(2i+1)

或： ki >= k(2i) 且 ki >= k(2i+1)

把此序列对应的二维数组看成一个完全二叉树。那么堆的含义就是：\*\*完全二叉树中任何一个非叶子节点的值均不大于（或不小于）其左，右孩子节点的值。\*\*由上述性质可知大顶堆的堆顶的关键字肯定是所有关键字中最大的，小顶堆的堆顶的关键字是所有关键字中最小的。因此我们可使用大顶堆进行升序排序, 使用小顶堆进行降序排序。

1. 基本思想

此处以大顶堆为例，堆排序的过程就是将待排序的序列构造成一个堆，选出堆中最大的移走，再把剩余的元素调整成堆，找出最大的再移走，重复直至有序。

1. 算法描述

①. 先将初始序列K[1..n]建成一个大顶堆, 那么此时第一个元素K1最大, 此堆为初始的无序区.

②. 再将关键字最大的记录K1 (即堆顶, 第一个元素)和无序区的最后一个记录 Kn 交换, 由此得到新的无序区K[1..n-1]和有序区K[n], 且满足K[1..n-1].keys <= K[n].key

③. 交换K1 和 Kn 后, 堆顶可能违反堆性质, 因此需将K[1..n-1]调整为堆. 然后重复步骤②, 直到无序区只有一个元素时停止.

1. 代码实现

从算法描述来看，堆排序需要两个过程，一是建立堆，二是堆顶与堆的最后一个元素交换位置。所以堆排序有两个函数组成。一是建堆函数，二是反复调用建堆函数以选择出剩余未排元素中最大的数来实现排序的函数。

操作定义：

最大堆调整（Max\_Heapify）：将堆的末端子节点作调整，使得子节点永远小于父节点

创建最大堆（Build\_Max\_Heap）：将堆所有数据重新排序

堆排序（HeapSort）：移除位在第一个数据的根节点，并做最大堆调整的递归运算

对于堆节点的访问：

父节点i的左子节点在位置：(2\*i+1);

父节点i的右子节点在位置：(2\*i+2);

子节点i的父节点在位置：floor((i-1)/2);

/\*\*

\* 堆排序

\*

\* 1. 先将初始序列K[1..n]建成一个大顶堆, 那么此时第一个元素K1最大, 此堆为初始的无序区.

\* 2. 再将关键字最大的记录K1 (即堆顶, 第一个元素)和无序区的最后一个记录 Kn 交换, 由此得到新的无序区K[1..n−1]和有序区K[n], 且满足K[1..n−1].keys⩽K[n].key

\* 3. 交换K1 和 Kn 后, 堆顶可能违反堆性质, 因此需将K[1..n−1]调整为堆. 然后重复步骤②, 直到无序区只有一个元素时停止.

\* @param arr 待排序数组

\*/

public static void heapSort(int[] arr){

for(int i = arr.length; i > 0; i--){

max\_heapify(arr, i);

int temp = arr[0]; //堆顶元素(第一个元素)与Kn交换

arr[0] = arr[i-1];

arr[i-1] = temp;

}

}

private static void max\_heapify(int[] arr, int limit){

if(arr.length <= 0 || arr.length < limit) return;

int parentIdx = limit / 2;

for(; parentIdx >= 0; parentIdx--){

if(parentIdx \* 2 >= limit){

continue;

}

int left = parentIdx \* 2; //左子节点位置

int right = (left + 1) >= limit ? left : (left + 1); //右子节点位置，如果没有右节点，默认为左节点位置

int maxChildId = arr[left] >= arr[right] ? left : right;

if(arr[maxChildId] > arr[parentIdx]){ //交换父节点与左右子节点中的最大值

int temp = arr[parentIdx];

arr[parentIdx] = arr[maxChildId];

arr[maxChildId] = temp;

}

}

System.out.println("Max\_Heapify: " + Arrays.toString(arr));

}

Note：x>>1 是位运算中的右移运算, 表示右移一位, 等同于x除以2再取整, 即 x>>1 == Math.floor(x/2)

①. 建立堆的过程, 从length/2 一直处理到0, 时间复杂度为O(n); ②. 调整堆的过程是沿着堆的父子节点进行调整, 执行次数为堆的深度, 时间复杂度为O(lgn); ③. 堆排序的过程由n次第②步完成, 时间复杂度为O(nlgn).

| **平均时间复杂度** | **最好情况** | **最坏情况** | **空间复杂度** |
| --- | --- | --- | --- |
| O(nlog2n) | O(nlog2n) | O(nlog2n) | O(1) |

### 1.1.3 快速排序(Quick Sort)

**典型的分治思想**

1. **基本思想**

**挖坑填数+分治法**

首先选一个轴值(pivot，也有叫基准的)，通过一趟排序将待排记录分隔成独立的两部分，其中一部分记录的关键字均比另一部分的关键字小，则可分别对这两部分记录继续进行排序，以达到整个序列有序。

1. 算法描述

快速排序使用分治策略来把一个序列（list）分为两个子序列（sub-lists）。步骤为：

①. 从数列中挑出一个元素，称为"基准"（pivot）。

②. 重新排序数列，所有比基准值小的元素摆放在基准前面，所有比基准值大的元素摆在基准后面（相同的数可以到任一边）。在这个分区结束之后，该基准就处于数列的中间位置。这个称为分区（partition）操作。

③. 递归地（recursively）把小于基准值元素的子数列和大于基准值元素的子数列排序。

递归到最底部时，数列的大小是零或一，也就是已经排序好了。这个算法一定会结束，因为在每次的迭代（iteration）中，它至少会把一个元素摆到它最后的位置去。

1. 代码实现

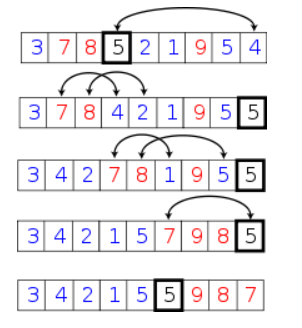
用伪代码描述如下：

①. i = L; j = R; 将基准数挖出形成第一个坑a[i]。

②．j--，由后向前找比它小的数，找到后挖出此数填前一个坑a[i]中。

③．i++，由前向后找比它大的数，找到后也挖出此数填到前一个坑a[j]中。

④．再重复执行②，③二步，直到i==j，将基准数填入a[i]中



/\*\*

\* 快速排序（递归）

\*

\* ①. 从数列中挑出一个元素，称为"基准"（pivot）。

\* ②. 重新排序数列，所有比基准值小的元素摆放在基准前面，所有比基准值大的元素摆在基准后面（相同的数可以到任一边）。在这个分区结束之后，该基准就处于数列的中间位置。这个称为分区（partition）操作。

\* ③. 递归地（recursively）把小于基准值元素的子数列和大于基准值元素的子数列排序。

\* @param arr 待排序数组

\* @param low 左边界

\* @param high 右边界

\*/

public static void quickSort(int[] arr, int low, int high){

if(arr.length <= 0) return;

if(low >= high) return;

int left = low;

int right = high;

int temp = arr[left]; //挖坑1：保存基准的值

while (left < right){

while(left < right && arr[right] >= temp){ //坑2：从后向前找到比基准小的元素，插入到基准位置坑1中

right--;

}

arr[left] = arr[right];

while(left < right && arr[left] <= temp){ //坑3：从前往后找到比基准大的元素，放到刚才挖的坑2中

left++;

}

arr[right] = arr[left];

}

arr[left] = temp; //基准值填补到坑3中，准备分治递归快排

System.out.println("Sorting: " + Arrays.toString(arr));

quickSort(arr, low, left-1);

quickSort(arr, left+1, high);

}

上面是递归版的快速排序：通过把基准temp插入到合适的位置来实现分治，并递归地对分治后的两个划分继续快排。

非递归操作：

因为递归的本质是栈，所以我们非递归实现的过程中，可以借助栈来保存中间变量就可以实现非递归了。在这里中间变量也就是通过Pritation函数划分区间之后分成左右两部分的首尾指针，只需要保存这两部分的首尾指针即可。

/\*\*

\* 快速排序（非递归）

\*

\* ①. 从数列中挑出一个元素，称为"基准"（pivot）。

\* ②. 重新排序数列，所有比基准值小的元素摆放在基准前面，所有比基准值大的元素摆在基准后面（相同的数可以到任一边）。在这个分区结束之后，该基准就处于数列的中间位置。这个称为分区（partition）操作。

\* ③. 把分区之后两个区间的边界（low和high）压入栈保存，并循环①、②步骤

\* @param arr 待排序数组

\*/

public static void quickSortByStack(int[] arr){

if(arr.length <= 0) return;

Stack<Integer> stack = new Stack<Integer>();

//初始状态的左右指针入栈

stack.push(0);

stack.push(arr.length - 1);

while(!stack.isEmpty()){

int high = stack.pop(); //出栈进行划分

int low = stack.pop();

int pivotIdx = partition(arr, low, high);

//保存中间变量

if(pivotIdx > low) {

stack.push(low);

stack.push(pivotIdx - 1);

}

if(pivotIdx < high && pivotIdx >= 0){

stack.push(pivotIdx + 1);

stack.push(high);

}

}

}

private static int partition(int[] arr, int low, int high){

if(arr.length <= 0) return -1;

if(low >= high) return -1;

int l = low;

int r = high;

int pivot = arr[l]; //挖坑1：保存基准的值

while(l < r){

while(l < r && arr[r] >= pivot){ //坑2：从后向前找到比基准小的元素，插入到基准位置坑1中

r--;

}

arr[l] = arr[r];

while(l < r && arr[l] <= pivot){ //坑3：从前往后找到比基准大的元素，放到刚才挖的坑2中

l++;

}

arr[r] = arr[l];

}

arr[l] = pivot; //基准值填补到坑3中，准备分治递归快排

return l;

}

快速排序是通常被认为在同数量级（O(nlog2n)）的排序方法中平均性能最好的。但若初始序列按关键码有序或基本有序时，快排序反而蜕化为冒泡排序。为改进之，通常以“三者取中法”来选取基准记录，即将排序区间的两个端点与中点三个记录关键码居中的调整为支点记录。快速排序是一个不稳定的排序方法。

算法复杂度：

| **平均时间复杂度** | **最好情况** | **最坏情况** | **空间复杂度** |
| --- | --- | --- | --- |
| O(nlog₂n) | O(nlog₂n) | O(n²) | O(1)（原地分区递归版） |

快速排序排序效率非常高。 虽然它运行最糟糕时将达到O(n²)的时间复杂度, 但通常平均来看, 它的时间复杂为O(nlogn), 比同样为O(nlogn)时间复杂度的归并排序还要快. 快速排序似乎更偏爱乱序的数列, 越是乱序的数列, 它相比其他排序而言, 相对效率更高.

Note: 同选择排序相似, 快速排序每次交换的元素都有可能不是相邻的, 因此它有可能打破原来值为相同的元素之间的顺序. 因此, 快速排序并不稳定.

### 1.1.4 归并排序(Merging Sort)

1）**基本思想**

**归并排序算法是将两个（或两个以上）有序表合并成一个新的有序表，即把待排序序列分为若干个子序列，每个子序列是有序的。然后再把有序子序列合并为整体有序序列。**

**2）算法描述**

**归并排序可通过两种方式实现：**

**自上而下的递归**

**自下而上的迭代**

1. **递归法（以序列有n个元素为例）**

①. 将序列每相邻两个数字进行归并操作，形成 floor(n/2)个序列，排序后每个序列包含两个元素；  
②. 将上述序列再次归并，形成 floor(n/4)个序列，每个序列包含四个元素；  
③. 重复步骤②，直到所有元素排序完毕。

1. **迭代法**

①. 申请空间，使其大小为两个已经排序序列之和，该空间用来存放合并后的序列   
②. 设定两个指针，最初位置分别为两个已经排序序列的起始位置   
③. 比较两个指针所指向的元素，选择相对小的元素放入到合并空间，并移动指针到下一位置   
④. 重复步骤③直到某一指针到达序列尾   
⑤. 将另一序列剩下的所有元素直接复制到合并序列尾

3）代码实现

**分解：将序列每次折半拆分**

**合并：将划分后的序列段两两排序合并**

**如何分解？**

**在这里，我们采用递归的方法，首先将待排序列分成A,B两组；然后重复对A、B序列 分组；直到分组后组内只有一个元素，此时我们认为组内所有元素有序，则分组结束。**

**如何合并？**

**L[first...mid]为第一段，L[mid+1...last]为第二段，并且两端已经有序，现在我们要将两端合成达到L[first...last]并且也有序。**

**首先依次从第一段与第二段中取出元素比较，将较小的元素赋值给temp[]**

**重复执行上一步，当某一段赋值结束，则将另一段剩下的元素赋值给temp[]**

**此时将temp[]中的元素复制给L[]，则得到的L[first...last]有序**

**递归算法**

/\*\*

\* 归并排序（递归）

\*

\* ①. 将序列每相邻两个数字进行归并操作，形成 floor(n/2)个序列，排序后每个序列包含两个元素；

\* ②. 将上述序列再次归并，形成 floor(n/4)个序列，每个序列包含四个元素；

\* ③. 重复步骤②，直到所有元素排序完毕。

\* @param arr 待排序数组

\*/

public static int[] mergingSort(int[] arr){

if(arr.length <= 1) return arr;

int num = arr.length >> 1;

int[] leftArr = Arrays.copyOfRange(arr, 0, num);

int[] rightArr = Arrays.copyOfRange(arr, num, arr.length);

System.out.println("split two array: " + Arrays.toString(leftArr) + " And " + Arrays.toString(rightArr));

return mergeTwoArray(mergingSort(leftArr), mergingSort(rightArr)); //不断拆分为最小单元，再排序合并

}

private static int[] mergeTwoArray(int[] arr1, int[] arr2){

int i = 0, j = 0, k = 0;

int[] result = new int[arr1.length + arr2.length]; //申请额外的空间存储合并之后的数组

while(i < arr1.length && j < arr2.length){ //选取两个序列中的较小值放入新数组

if(arr1[i] <= arr2[j]){

result[k++] = arr1[i++];

}else{

result[k++] = arr2[j++];

}

}

while(i < arr1.length){ //序列1中多余的元素移入新数组

result[k++] = arr1[i++];

}

while(j < arr2.length){ //序列2中多余的元素移入新数组

result[k++] = arr2[j++];

}

System.out.println("Merging: " + Arrays.toString(result));

return result;

}

**长度为n的数组, 最终会调用mergeSort函数2n-1次。通过自上而下的递归实现的归并排序, 将存在堆栈溢出的风险。**

**算法复杂度、**

| **平均时间复杂度** | **最好情况** | **最坏情况** | **空间复杂度** |
| --- | --- | --- | --- |
| O(nlog₂n) | O(nlog₂n) | O(nlog₂n) | O(n) |

**从效率上看，归并排序可算是排序算法中的”佼佼者”. 假设数组长度为n，那么拆分数组共需logn，, 又每步都是一个普通的合并子数组的过程， 时间复杂度为O(n)， 故其综合时间复杂度为O(nlogn)。另一方面， 归并排序多次递归过程中拆分的子数组需要保存在内存空间， 其空间复杂度为O(n)。**

**Note：和选择排序一样，归并排序的性能不受输入数据的影响，但表现比选择排序好的多，因为始终都是O(n log n）的时间复杂度。代价是需要额外的内存空间。**

## 1.2 非基于比较排序

### 1.2.1 基数排序（Radix Sort）

**算法复杂度**

| **平均时间复杂度** | **最好情况** | **最坏情况** | **空间复杂度** |
| --- | --- | --- | --- |
| O(d\*(n+r)) | O(d\*(n+r)) | O(d\*(n+r)) | O(n+r) |

其中，**d 为位数，r 为基数，n 为原数组个数**。在基数排序中，因为没有比较操作，所以在复杂上，最好的情况与最坏的情况在时间上是一致的，均为 O(d\*(n + r))。

Note：基数排序更适合用于对时间, 字符串等这些**整体权值未知**的数据进行排序；基数排序不改变相同元素之间的相对顺序，因此它是稳定的排序算法。

### 1.2.2 基数排序 vs 计数排序 vs 桶排序

这三种排序算法都利用了桶的概念，但对桶的使用方法上有明显差异：

1. 基数排序：根据键值的每位数字来分配桶
2. 计数排序：每个桶只存储单一键值
3. 桶排序：每个桶存储一定范围的数值

# 二、总结



## 2.1 从时间复杂度来说

(1). 平方阶**O(n²)**排序：**各类简单排序：直接插入、直接选择和冒泡排序；**

(2). 线性对数阶**O(nlog₂n)**排序：**快速排序、堆排序和归并排序**；

(3). O(n1+§))排序，§是介于0和1之间的常数：**希尔排序**

(4). 线性阶O(n)排序：**基数排序**，此外还有**桶**、**箱排序**。

基数排序只适用于有基数的情况，而基于比较的排序适用范围就广得多。另一方面是内存上的考虑。作为一种通用的排序方法，最好不要带来意料之外的内存开销，所以各语言的默认实现都没有用基数排序，但是不能否认基数排序在各领域的应用。

## 2.2 时间复杂度极限

**当被排序的数有一些性质的时候**（比如是整数，比如有一定的范围），排序算法的复杂度是可以小于O(nlgn)的。比如：

1. 计数排序 复杂度O( k+n) 要求：被排序的数是0~k范围内的整数
2. 基数排序 复杂度O( d(k+n) ) 要求：d位数，每个数位有k个取值
3. 桶排序 复杂度 O( n ) （平均） 要求：被排序数在某个范围内，并且服从均匀分布

**但是，当被排序的数不具有任何性质的时候，一般使用基于比较的排序算法，而基于比较的排序算法时间复杂度的下限必须是O(nlgn)。**

# 说明

* 当原表有序或基本有序时，直接插入排序和冒泡排序将大大减少比较次数和移动记录的次数，时间复杂度可降至O（n）；
* 而快速排序则相反，当原表基本有序时，将蜕化为冒泡排序，时间复杂度提高为O（n2）；
* 原表是否有序，对简单选择排序、堆排序、归并排序和基数排序的时间复杂度影响不大。

**常用排序算法**

****

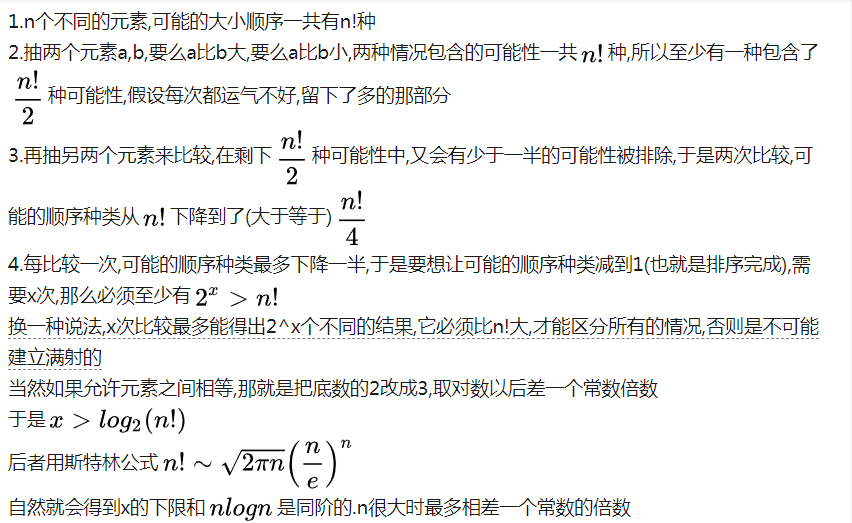
**Question:**

**很多高效排序算法的代价为nlogn，这是排序算法的极限吗？**

**Answer:**

**这是基于比较排序算法的极限；**

**分析：**



**需要注意的是：快速排序、堆排序和归并排序，它们之间的差别在于常数的倍数上，都没有达到理论上的最优！**