在计算机科学所使用的排序算法通常被分类为：

* 计算的 ****时间复杂度****（最差、平均、和最好性能），依据列表（list）的大小(n)。一般而言，好的性能是O(n log n)，且坏的性能是O(n^2)。对于一个排序理想的性能是O(n)。仅使用一个抽象关键比较运算的排序算法总平均上总是至少需要O(n log n)。
* ****存储器使用量****（以及其他电脑资源的使用）
* ****稳定性****：稳定排序算法会让原本有相等键值的纪录维持相对次序。也就是如果一个排序算法是稳定的，当有两个相等键值的纪录R和S，且在原本的列表中R出现在S之前，在排序过的列表中R也将会是在S之前。
* ****依据排序的方法****：插入、交换、选择、合并等等。

依据排序的方法分类的三种排序算法:

## 冒泡排序

冒泡排序对一个需要进行排序的数组进行以下操作:

1. 比较第一项和第二项;
2. 如果第一项应该排在第二项之后, 那么两者交换顺序;
3. 比较第二项和第三项;
4. 如果第二项应该排在第三项之后, 那么两者交换顺序;
5. 以此类推直到完成排序;

****实例说明:****

将数组[3, 2, 4, 5, 1]以从小到大的顺序进行排序:

1. 3应该在2之后, 因此交换, 得到[2, 3, 4, 5, 1];
2. 3, 4顺序不变, 4, 5也不变, 交换5, 1得到[2, 3, 4, 1, 5];
3. 第一次遍历结束, 数组中最后一项处于正确位置不会再有变化, 因此下一次遍历可以排除最后一项;
4. 开始第二次遍历, 最后结果为[2, 3, 1, 4, 5], 排除后两项进行下一次遍历;
5. 第三次遍历结果为[2, 1, 3, 4, 5];
6. 最后得到[1, 2, 3, 4, 5], 排序结束;

****代码实现:****

function **swap(items,** firstIndex, secondIndex){

var temp = items[firstIndex]*;*

items[firstIndex] = items[secondIndex]*;*

items[secondIndex] = temp*;*

}*;*

function **bubbleSort(items){** var len = items.length, i, **j,** stop*;*

for (i = 0*; i < len; i++){*

for (**j** = 0, stop = len-i*; j < stop; j++){*

if (items[**j]** > items[**j+1]){** **swap(items, j, j+1);** }

}

}

return items*;*

}

外层的循环决定需要进行多少次遍历, 内层的循环负责数组内各项的比较, 还通过外层循环的次数和数组长度决定何时停止比较.

冒泡排序极其低效, 因为处理数据的步骤太多, 对于数组中的每n项, 都需要n^2次操作来实现该算法(实际比n^2略小, 但可以忽略, 具体原因见⤵️), 即时间复杂度为O(n^2).

对于含有n个元素的数组, 需要进行(n-1)+(n-2)+...+1次操作, 而(n-1)+(n-2)+...+1 = n(n-1)/2 = n^2/2 - n/2, 如果n趋于无限大, 那么n/2的大小对于整个算式的结果影响可以忽略, 因此最终的时间复杂度用O(n^2)表示

## 选择排序

选择排序对一个需要进行排序的数组进行以下操作:

1. 假定数组中的第一项为最小值(min);
2. 比较第一项和第二项的值;
3. 若第二项比第一项小, 则假定第二项为最小值;
4. 以此类推直到排序完成.

****实例说明:****

将数组["b", "a", "d", "c", "e"]以字母a-z的顺序进行排序:

1. 假定数组中第一项"b"(index0)为min;
2. 比较第二项"a"与第一项"b", 因"a"应在"b"之前的顺序, 故"a"(index1)为min;
3. 然后将min与后面几项比较, 由于"a"就是最小值, 因此min确定在index1的位置;
4. 第一次遍历结束后, 将假定的min(index0), 与真实的min(index1)进行比较, 真实的min应该在index0的位置, 因此将两者交换, 第一次遍历交换之后的结果为["a", "b", "d", "c", "e"];
5. 然后开始第二次遍历, 遍历从第二项(index1的位置)开始, 这次假定第二项为最小值, 将第二项与之后几项逐个比较, 因为"b"就在应该存在的位置, 所以不需要进行交换, 这次遍历之后的结果为"a", "b", "d", "c", "e"];
6. 之后开始第三次遍历, "c"应为这次遍历的最小值, 交换index2("d"), index3("c")位置, 最后结果为["a", "b", "c", "d", "e"];
7. 最后一次遍历, 所有元素在应有位置, 不需要进行交换.

****代码实现:****

function swap(items, firstIndex, secondIndex){

var temp = items[firstIndex];

items[firstIndex] = items[secondIndex];

items[secondIndex] = temp;

};

function selectionSort(){

let items = [...document.querySelectorAll('.num-queue span')].map(num => +num.textContent);

let len = items.length, min;

**for** (i = 0; i < len; i++){

min = i;

**for**(j = i + 1; j < len; j++){

**if**(items[j] < items[min]){

min = j;

}

}

**if**(i != min){

swap(items, i, min);

}

}

return items;

};

外层循环决定每次遍历的初始位置, 从数组的第一项开始直到最后一项. 内层循环决定哪一项元素被比较.

选择排序的时间复杂度为O(n^2).

## 插入排序

与上述两种排序算法不同, 插入排序是稳定排序算法(stable sort algorithm), 稳定排序算法指不改变列表中相同元素的位置, 冒泡排序和选择排序不是稳定排序算法, 因为排序过程中有可能会改变相同元素位置. 对简单的值(数字或字符串)排序时, 相同元素位置改变与否影响不是很大. 而当列表中的元素是对象, 根据对象的某个属性对列表进行排序时, 使用稳定排序算法就很有必要了.

一旦算法包含交换(swap)这个步骤, 就不可能是稳定的排序算法. 列表内元素不断交换, 无法保证先前的元素排列为止一直保持原样. 而插入排序的实现过程不包含交换, 而是提取某个元素将其插入数组中正确位置.

插入排序的实现是将一个数组分为两个部分, 一部分排序完成, 一部分未进行排序. 初始状态下整个数组属于未排序部分, 排序完成部分为空. 然后进行排序, 数组内的第一项被加入排序完成部分, 由于只有一项, 自然属于排序完成状态. 然后对未完成排序的余下部分的元素进行如下操作:

1. 如果这一项的值应该在排序完成部分最后一项元素之后, 保留这一项在原有位置开始下一步;
2. 如果这一项的值应该排在排序完成部分最后一项元素之前, 将这一项从未完成部分暂时移开, 将已完成部分的最后一项元素移后一个位置;
3. 被暂时移开的元素与已完成部分倒数第二项元素进行比较;
4. 如果被移除元素的值在最后一项与倒数第二项的值之间, 那么将其插入两者之间的位置, 否则继续与前面的元素比较, 将暂移出的元素放置已完成部分合适位置. 以此类推直到所有元素都被移至排序完成部分.

****实例说明:****

现在需要将数组var items = [5, 2, 6, 1, 3, 9];进行插入排序:

1. 5属于已完成部分, 余下元素为未完成部分. 接下来提取出2, 因为5比2大, 于是5被移至靠右一个位置, 覆盖2, 占用2原本存在的位置. 这样本来存放5的位置(已完成部分的首个位置)就被空出, 而2在比5小, 因此将2置于这个位置, 此时结果为[2, 5, 6, 1, 3, 9];
2. 接下来提取出6, 因为6比5大, 所以不操作提取出1, 1与已完成部分各个元素(2, 5, 6)进行比较, 应该在2之前, 因此2, 5, 6各向右移一位, 1置于已完成部分首位, 此时结果为[1, 2, 5, 6, 3, 9];
3. 对余下未完成元素进行类似操作, 最后得出结果[1, 2, 3, 5, 6, 9];

****代码实现:****

function **insertionSort(items)** {

let len = items.length, value, i, **j;** for (i = 0*; i < len; i++) {*

value = items[i]*;*

for (**j** = i-1*; j > -1 && items[j] > value; j--) {*

items[**j+1]** = items[**j];** }

items[**j+1]** = value*;*

}

return items*;*

}*;*

外层循环的遍历顺序是从数组的第一位到最后一位, 内层循环的遍历则是从后往前, 内层循环同时负责元素的移位.

插入排序的时间复杂度为O(n^2)

****以上三种排序算法都十分低效, 因此实际应用中不要使用这三种算法, 遇到需要排序的问题, 应该首先使用JavaScript内置的方法**Array.prototype.sort()**;****