# 1算法复杂度

## 1.1认识时间复杂度

时间复杂度为一个算法流程中，常数操作数量的指标，这个指标叫做O，big O。具体为，如果常数操作数量的表达式中，只要高阶项，不要低阶项，也不要高阶项系数之后，剩下的部分记为f(N)，那么该算法的时间复杂度为O(f(N))，用表示。

## 1.2冒泡排序

即大数依次往下沉。需要操作次。时间复杂度，额外空间复杂度（与数组长度无关），实现可以做到稳定性（是指一个数组在排完序后值相等的数顺序不会发生变化）。异或——无进位相加；

**function** bubbleSort(arr){  
 **if**(arr.length<2){  
 **return** arr;  
 }  
 **for** (**var** e=arr.length-1;e>0;e--){  
 **for** (**var** i = 0; i < e; i++){  
 **if**(arr[i]>arr[i+1]){  
 swap(arr,i,i+1);  
 }  
 }  
 }  
 **return** arr;  
}  
**function** swap(arr,i,j){  
 /\*只有在i与j不相等时才能用，且arr内值均为整数，交换两数  
 arr[i]=arr[i]^arr[j];  
 arr[j]=arr[i]^arr[j];  
 arr[i]=arr[i]^arr[j];\*/  
 **var** tmp=arr[i];  
 arr[j]=tmp;  
 arr[i]=arr[j];  
}  
console.log(bubbleSort([2,5,1,6,3]));

## 1.3插入排序

——对比第n个数字与前n-1个排序好的数字，将第n个依次与第n-1、n-2…比较，若小于，则往上浮。将这第n个数字插入到左边排序好的数字其中，类似于扑克牌。时间复杂度，额外空间复杂度，实现可以做到稳定性。

**function** insertSort(arr) {  
 **if** (arr.length < 2) {  
 **return** arr;  
 }  
 **for** (**var** i = 0, l = arr.length ; i < l; i++) {//i为要插入的数  
 **for** (**var** j = i - 1; j >= 0 && arr[j] > arr[j + 1]; j--) {  
 swap(arr, j, j + 1);  
 }  
 }  
 **return** arr;  
}  
**function** swap(arr, i, j) {  
 **var** tmp = arr[i];  
 arr[i] = arr[j];  
 arr[j] = tmp;  
}  
console.log(insertSort([2, 5, 1, 6, 3]));

## 1.4选择排序

——依次选出最小数放在第1,2,3…n个。时间复杂度，额外空间复杂度，实现可以做到稳定性。

**function** selectioSort(arr) {  
 **if** (arr.length < 2) {  
 **return** arr;  
 }  
 **for** (**var** i = 0, l = arr.length; i < l; i++) {  
 **var** minIndex = i; //最小数索引  
 **for** (**var** j = i + 1; j < l; j++) {  
 minIndex = arr[j] < arr[minIndex] ? j : minIndex;  
 }  
 swap(arr, i, minIndex);  
 }  
 **return** arr;  
}  
**function** swap(arr, i, j) {  
 **var** tmp = arr[i];  
 arr[i] = arr[j];  
 arr[j] = tmp;  
}  
console.log(selectionSort([2, 5, 1, 6, 3]));

## 1.5随机快速排序

——随机选一个数，小于它的放左边，等于的放中间，大于的放右边。然后在两边继续随机选数排序。

时间复杂度，额外空间复杂度，常规实现做不到稳定性。

注意：

1. 快速排序中，额外空间复杂度最低为
2. 快速排序，可以做到稳定性的实现，但是非常难，不需要掌握
3. 荷兰国旗问题的实现，和快速排序中的改进。

**function** randomQuickSort(arr) {  
 **if** (arr.length < 2) {  
 **return** arr;  
 }  
 arr=quickSort(arr,0,arr.length-1);  
 **return** arr;  
}  
**function** quickSort(arr, l, r) {  
 **if** (l < r) {  
 swap(arr, l + Math.round(Math.random() \* (r - l + 1)), r);//随机选择一个数放在最后  
 **var** p = partition(arr, l, r);  
 quickSort(arr, l, p[0]); //小于区排序  
 quickSort(arr, p[1] + 1, r); //大于区排序  
 }  
 **return** arr;  
}  
**function** partition(arr, l, r) {  
 **var** less = l - 1; //小于区域最后一个  
 **var** more = r; //大于区域最前面一个  
 **while** (l < more) {  
 **if** (arr[l] < arr[r]) {//小于随机数时，该数与小于区边界后一个交换  
 swap(arr, ++less, l++);  
 } **else if** (arr[l] > arr[r]) {//大于随机数时，该数与大于区前一个数交换  
 swap(arr, --more, l);  
 } **else** {//等于时跳下一个  
 l++;  
 }  
 }  
 swap(arr, more, r); //将选择的随机数放到大于区第一个  
 **return** [less, more]; //返回小于区和大于区的边界  
}  
**function** swap(arr, i, j) {  
 **var** tmp = arr[i];  
 arr[i] = arr[j];  
 arr[j] = tmp;  
}  
console.log(randomQuickSort([2, 5, 1, 6, 3]));

## 1.6归并排序

——大数组依次对半拆分至单个数再往上合并。

时间复杂度，额外空间复杂度，实现可以做到稳定性。

注意：

1. 库函数中排序的实现是综合排序，比如插入+快速；比如为了稳定性，排序算法往往是快排+堆排序；
2. 归并排序和快速排序，都一定存在非递归的实现；
3. 归并排序，存在额外空间复杂度O(1)的实现（内部缓存法），但是非常难，你不需要掌握；
4. 归并排序的扩展，小和问题，逆序对。

**function** bigMergeSort(arr) {  
 **if** (arr.length < 2) {  
 **return** arr;  
 }  
 mergeSort(arr, 0, arr.length - 1);  
 **return** arr;  
}  
**function** mergeSort(arr, l, r) {  
 **if** (l === r) {  
 **return** arr;  
 }  
 //取中位数，如果直接用(l+r)/2可能会溢出，而右移相当于除2，而且比除法速度快。  
 **var** mid = l + ((r - l) >> 1);  
 mergeSort(arr, l, mid);  
 mergeSort(arr, mid + 1, r);  
 merge(arr, l, mid, r); //排序好的两半合并  
}  
**function** merge(arr, l, m, r) {  
 **var** tmp = **new** Array(r - l + 1);  
 **var** i = 0, p1 = l, p2 = m + 1;  
 **while** (p1 <= m && p2 <= r) {  
 tmp[i++] = arr[p1] < arr[p2] ? arr[p1++] : arr[p2++];  
 }  
 **while** (p1 <= m) {  
 tmp[i++] = arr[p1++];  
 }  
 **while** (p2 <= r) {  
 tmp[i++] = arr[p2++];  
 }  
 **for** (i = 0; i < tmp.length; i++) {  
 arr[l + i] = tmp[i];  
 }  
 **return** arr;  
}  
console.log(bigMergeSort([2, 5, 1, 6, 3]));

小和问题：

在随机元素，随机数组大小的数组中，找出左边比右边元素小的所有元素之和。

**function** smallSum(arr) {  
 **if** (arr.length < 2) {  
 **return** 0;  
 }  
 **return** mergeSort(arr, 0, arr.length - 1);  
}  
**function** mergeSort(arr, l, r) {  
 **if** (l === r) {  
 **return** 0;  
 }  
 //取中位数，如果直接用(l+r)/2可能会溢出，而右移相当于除2，而且比除法速度快。  
 **var** mid = l + ((r - l) >> 1);  
 **return** mergeSort(arr, l, mid) + mergeSort(arr, mid + 1, r) + merge(arr, l, mid, r);  
}  
**function** merge(arr, l, m, r) {  
 **var** tmp = **new** Array(r - l + 1);  
 **var** i = 0, p1 = l, p2 = m + 1, res = 0;  
 **while** (p1 <= m && p2 <= r) {  
 res += arr[p1] < arr[p2] ? (r - p2 + 1) \* arr[p1] : 0; //提取小和  
 tmp[i++] = arr[p1] < arr[p2] ? arr[p1++] : arr[p2++];  
 }  
 **while** (p1 <= m) {  
 tmp[i++] = arr[p1++];  
 }  
 **while** (p2 <= r) {  
 tmp[i++] = arr[p2++];  
 }  
 **for** (i = 0; i < tmp.length; i++) {  
 arr[l + i] = tmp[i];  
 }  
 **return** res;  
}  
console.log(smallSum([2, 5, 1, 6, 3]));

## 1.7堆排序

时间复杂度，额外空间复杂度，实现做不到稳定性。

关键步骤：heapInsert，heapify，堆的扩大和缩小操作。

注意：

1. 堆排序中，建立堆的操作为O(n)；
2. 堆排序的核心数据结构：堆，也可以说是优先级队列；

堆：完全二叉树结构

大根堆：任何一个节点都是下面整棵树的最大的值。

一个节点i，父节点为(i-1)/2，左孩子为2\*i+1，右孩子为2\*i+2

先建大根堆，然后堆顶最大值与堆尾互换，新的堆顶下沉，继续建大根堆；

**function** heapSort(arr) {  
 **if** (arr.length < 2) {  
 **return** arr;  
 }  
 **for** (**var** i = 0, l = arr.length; i < l; i++) {  
 heapInsert(arr, i);  
 }  
 **var** size=arr.length;  
 swap(arr,0,--size);  
 **while** (size>0){  
 heapify(arr,0,size);  
 swap(arr,0,--size)  
 }  
 **return** arr;  
}  
//建立大根堆  
**function** heapInsert(arr, index) {  
 **while** (arr[index] > arr[(index - 1) / 2]) {  
 swap(arr, index, (index - 1) / 2);  
 index = (index - 1) / 2;  
 }  
}  
//根顶下沉  
**function** heapify(arr, index, size) {//下沉操作  
 **var** left = index \* 2 + 1;  
 **while** (left < size) {//是否有后代  
 //找出较大孩子序号  
 **var** largest = left + 1 < size && arr[left + 1] > arr[left] ? left + 1 : left;  
 largest = arr[largest] > arr[index] ? largest : index;//我以及我孩子中的最大值序号  
 **if** (largest === index) {  
 **break**;  
 }  
 swap(arr, largest, index);  
 index = largest;  
 left = index \* 2 + 1;  
 }  
}  
//互换函数  
**function** swap(arr, i, j) {  
 **var** tmp = arr[i];  
 arr[i] = arr[j];  
 arr[j] = tmp;  
}  
console.log(heapSort([2, 5, 1, 6, 3]));

## 1.8桶排序与基数排序

时间复杂度O(n)，额外空间复杂度O(n)，实现做到稳定性。

注意:

1. 桶排序的扩展，排序后的最大相邻数差值问题

解法：数组长度为N，准备N+1个桶，将数组的最大值减最小值的范围均分N+1份，每一份代表桶的范围，依次把数组内的数放入桶内部，求出非空桶内的最小值与前一个非空桶的最大值的差值，其中最大的就为最大相邻数差值。例如数组[1,3,5,11]，最大值减最小值范围是10，则5个桶的范围是[1~3,3~5,5~7,7~9,9~11]。

**function** maxGap(arr) {  
 **if** (arr.length < 2) {  
 **return** 0;  
 }  
 **var** len = arr.length;  
 **var** max = Number.MIN\_VALUE;  
 **var** min = Number.MAX\_VALUE;  
 **for** (**var** n = 0; n < len; n++) {  
 min = Math.min(min, arr[n]);  
 max = Math.max(max, arr[n]);  
 }  
 **if** (max === min) {  
 **return** 0;  
 }  
 **var** hasNum = **new** Array(len + 1);//桶内是否有数数组  
 **var** maxs = **new** Array(len + 1);//桶内最大值数组  
 **var** mins = **new** Array(len + 1);//桶内最小值数组  
 **var** bid = 0;  
 **for** (**var** i = 0; i < len; i++) {  
 bid = bucket(arr[i], len, min, max);  
 //更新桶信息  
 mins[bid] = hasNum[bid] ? Math.min(mins[bid], arr[i]) : arr[i];  
 maxs[bid] = hasNum[bid] ? Math.max(maxs[bid], arr[i]) : arr[i];  
 hasNum[bid] = **true**;  
 }  
 **var** res = 0;  
 **var** lastMax = maxs[0]; //上一个桶的最大值  
 **for** (**var** j = 1; j <= len; j++) {  
 **if** (hasNum[j]) {  
 res = Math.max(res, mins[j] - lastMax); //取非空桶与上一个桶的差值中的最大值。  
 lastMax = maxs[j];  
 }  
 }  
 **return** res;  
  
}  
*/\*\*  
 \* 判断数字进第几号桶  
 \** ***@param*** *num  
 \** ***@param*** *len  
 \** ***@param*** *min  
 \** ***@param*** *max  
 \** ***@returns*** *{Number}  
 \*/***function** bucket(num, len, min, max) {  
 **return** parseInt((num - min) \* len / (max - min));  
}  
console.log(maxGap([1, 3, 5, 11, 8, 4]));

1. 非基于比较的排序，对数据的位数和范围有限制。