# 排序算法

## 常见排序算法

### 稳定排序：

* 冒泡排序 — O(n²)
* 插入排序 — O(n²)
* 桶排序 — O(n); 需要 O(k) 额外空间
* 归并排序 — O(nlogn); 需要 O(n) 额外空间
* 二叉排序树排序 — O(n log n) 期望时间; O(n²)最坏时间; 需要 O(n) 额外空间
* 基数排序 — O(n·k); 需要 O(n) 额外空间

### 不稳定排序

* 选择排序 — O(n²)
* 希尔排序 — O(nlogn)
* 堆排序 — O(nlogn)
* 快速排序 — O(nlogn) 期望时间, O(n²) 最坏情况; 对于大的、乱数串行一般相信是最快的已知排序

## 交换排序

### 冒泡排序

它重复地走访过要排序的数列，一次比较两个元素，如果他们的顺序错误就把他们交换过来。走访数列的工作是重复地进行直到没有再需要交换，也就是说该数列已经排序完成。**冒泡排序总的平均时间复杂度为O(n^2)。冒泡排序是一种稳定排序算法。** - 比较相邻的元素。如果第一个比第二个大，就交换他们两个。 - 对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对。在这一点，最后的元素应该会是最大的数。 - 针对所有的元素重复以上的步骤，除了最后一个。 - 持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较。

void bubble\_sort(int a[], int n)  
{  
 int i, j, temp;  
 for (j = 0; j < n - 1; j++)  
 for (i = 0; i < n - 1 - j; i++)  
 {  
 if(a[i] > a[i + 1])  
 {  
 temp = a[i];  
 a[i] = a[i + 1];  
 a[i + 1] = temp;  
 }  
 }  
}

### 快速排序 [快速排序-百度百科](http://baike.baidu.com/link?url=hyQPClbJy1SYY4esOZe9kANDIDxOrKxiSfq0HZl8c5eut40dZS-fd1V0jubijSv7RAogwy6HaQ-B1HbRgHf1hq)

快速排序是一种 **不稳定** 的排序算法，平均时间复杂度为 **O(nlogn)**。**快速排序使用分治法（Divide and conquer）策略来把一个序列（list）分为两个子序列（sub-lists）。** 步骤为：

* 从数列中挑出一个元素，称为"基准"（pivot），
* 重新排序数列，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面（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。
* 递归地（recursive）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。

快排的时间花费主要在划分上，所以 - 最坏情况：时间复杂度为O(n^2)。因为最坏情况发生在每次划分过程产生的两个区间分别包含n-1个元素和1个元素的时候。 - 最好情况：每次划分选取的基准都是当前无序区的中值。如果每次划分过程产生的区间大小都为n/2，则快速排序法运行就快得多了。

public void sort(int[] arr, int low, int high) {  
 int l = low;  
 int h = high;  
 int povit = arr[low];  
  
 while (l < h) {  
 while (l < h && arr[h] >= povit)  
 h--;  
 if (l < h) {  
 arr[l] = arr[h];  
 l++;  
 }  
  
 while (l < h && arr[l] <= povit)  
 l++;  
  
 if (l < h) {  
 arr[h] = arr[l];  
 h--;  
 }  
 }  
  
 arr[l] = povit;  
  
 System.out.print("l=" + (l + 1) + ";h=" + (h + 1) + ";povit=" + povit + "\n");  
 System.out.println(Arrays.toString(arr));  
 if (l - 1 > low) sort(arr, low, l - 1);  
 if (h + 1 < high) sort(arr, h + 1, high);  
 }

#### 快排的优化

1. 当待排序序列的长度分割到一定大小后，使用插入排序。
2. 快排函数在函数尾部有两次递归操作，我们可以对其使用尾递归优化。优化后，可以缩减堆栈深度，由原来的O(n)缩减为O(logn)，将会提高性能。
3. 从左、中、右三个数中取中间值。

## 插入排序

### 直接插入排序

插入排序的基本操作就是将一个数据插入到已经排好序的有序数据中，从而得到一个新的、个数加一的有序数据，算法适用于少量数据的排序，**时间复杂度为O(n^2)。是稳定的排序方法。** **插入算法把要排序的数组分成两部分**：第一部分包含了这个数组的所有元素，但将最后一个元素除外（让数组多一个空间才有插入的位置），而第二部分就只包含这一个元素（即待插入元素）。在第一部分排序完成后，再将这个最后元素插入到已排好序的第一部分中。

void insert\_sort(int\* a, int len) {  
 for (int i = 1; i < len; ++i) {  
 int j = i - 1;  
 int temp = a[i];  
 while (j >= 0 && temp < a[j]) {  
 a[j + 1] = a[j];  
 j--;  
 }  
 a[j + 1] = temp;  
 }  
}

### [希尔排序](https://zh.wikipedia.org/wiki/%E5%B8%8C%E5%B0%94%E6%8E%92%E5%BA%8F)

也称缩小增量排序，是直接插入排序算法的一种更高效的改进版本。**希尔排序是非稳定排序算法。**

希尔排序是把记录按下标的一定增量分组，对每组使用直接插入排序算法排序；随着增量逐渐减少，每组包含的关键词越来越多，当增量减至1时，整个文件恰被分成一组，算法便终止。

void shell\_sort(int\* a, int len) {  
 int step = len / 2;  
 int temp;  
  
 while (step > 0) {  
 for (int i = step; i < len; ++i) {  
 temp = a[i];  
 int j = i - step;  
 while (j >= 0 && temp < a[j]) {  
 a[j + step] = a[j];  
 j -= step;  
 }  
 a[j + step] = temp;  
 }  
 step /= 2;  
 }  
}

## 选择排序

### 直接选择排序

首先在未排序序列中找到最小（大）元素，存放到排序序列的起始位置，然后，再从剩余未排序元素中继续寻找最小（大）元素，然后放到已排序序列的末尾。**实际适用的场合非常罕见。**

void selection\_sort(int arr[], int len) {  
 int i, j, min, temp;  
 for (i = 0; i < len - 1; i++) {  
 min = i;  
 for (j = i + 1; j < len; j++)  
 if (arr[min] > arr[j])  
 min = j;  
 temp = arr[min];  
 arr[min] = arr[i];  
 arr[i] = temp;  
 }  
}

### 堆排序

堆排序利用了大根堆（或小根堆）堆顶记录的关键字最大（或最小）这一特征，使得在当前无序区中选取最大（或最小）关键字的记录变得简单。

1. 将数组分为有序区和无序区，在无序区中建立最大堆
2. 将堆顶的数据与无序区末尾的数据交换
3. 从后往前，直到所有数据排序完成

public void heapSort(int[] nums) {  
 for (int i = nums.length - 1; i >= 0; i--) {  
 maxHeap(nums, 0, i);  
  
 swap(nums, 0, i);  
 }  
}  
  
public void maxHeap(int[] heap, int start, int end) {  
 if (start == end) {  
 return;  
 }  
  
 int parent = start;  
 int childLeft = start \* 2 + 1;  
 int childRight = childLeft + 1;  
  
 if (childLeft <= end) {  
 maxHeap(heap, childLeft, end);  
  
 if (heap[childLeft] > heap[parent]) {  
 swap(heap, parent, childLeft);  
 }  
 }  
  
 if (childRight <= end) {  
 maxHeap(heap, childRight, end);  
  
 if (heap[childRight] > heap[parent]) {  
 swap(heap, parent, childRight);  
 }  
 }  
}  
  
private void swap(int[] nums, int a, int b) {  
 int t = nums[a];  
 nums[a] = nums[b];  
 nums[b] = t;  
}

## 归并排序

归并排序采用分治的思想： - Divide：将n个元素平均划分为各含n/2个元素的子序列； - Conquer：递归的解决俩个规模为n/2的子问题； - Combine：合并俩个已排序的子序列。

性能：时间复杂度总是为O(NlogN)，空间复杂度也总为为O(N)，算法与初始序列无关，排序是稳定的。

public void mergeSort(int[] array, int start, int end, int[] temp) {  
 if (start >= end) {  
 return;  
 }  
  
 int mid = (start + end) / 2;  
  
 mergeSort(array, start, mid, temp);  
 mergeSort(array, mid + 1, end, temp);  
  
 int f = start, s = mid + 1;  
 int t = 0;  
 while (f <= mid && s <= end) {  
 if (array[f] < array[s]) {  
 temp[t++] = array[f++];  
 } else {  
 temp[t++] = array[s++];  
 }  
 }  
  
 while (f <= mid) {  
 temp[t++] = array[f++];  
 }  
  
 while (s <= end) {  
 temp[t++] = array[s++];  
 }  
  
 for (int i = 0, j = start; i < t; i++) {  
 array[j++] = temp[i];  
 }  
}

## 基数排序

对于有d个关键字时，可以分别按关键字进行排序。有俩种方法： - MSD：先从高位开始进行排序，在每个关键字上，可采用基数排序 - LSD：先从低位开始进行排序，在每个关键字上，可采用桶排序

即通过每个数的每位数字的大小来比较

//找出最大数字的位数  
int maxNum(int arr[], int len) {  
 int \_max = 0;  
  
 for (int i = 0; i < len; ++i) {  
 int d = 0;  
 int a = arr[i];  
  
 while (a) {  
 a /= 10;  
 d++;  
 }  
  
 if (\_max < d) {  
 \_max = d;  
 }  
 }  
 return \_max;  
}  
  
  
void radixSort(int \*arr, int len) {  
 int d = maxNum(arr, len);  
 int \*temp = new int[len];  
 int count[10];  
 int radix = 1;  
  
 for (int i = 0; i < d; ++i) {  
 for (int j = 0; j < 10; ++j) {  
 count[j] = 0;  
 }  
  
 for (int k = 0; k < len; ++k) {  
 count[(arr[k] / radix) % 10]++;  
 }  
  
 for (int l = 1; l < 10; ++l) {  
 count[l] += count[l - 1];  
 }  
  
 for (int m = 0; m < len; ++m) {  
 int index = (arr[m] / radix) % 10;  
 temp[count[index] - 1] = arr[m];  
 count[index]--;  
 }  
  
 for (int n = 0; n < len; ++n) {  
 arr[n] = temp[n];  
 }  
 radix \*= 10;  
  
 }  
  
 delete (temp);  
}

## 拓扑排序

在有向图中找拓扑序列的过程，就是拓扑排序。**拓扑序列常常用于判定图是否有环**。

* 从有向图中选择一个入度为0的结点，输出它。
* 将这个结点以及该结点出发的所有边从图中删除。
* 重复前两步，直到没有入度为0的点。

如果所有点都被输出，即存在一个拓扑序列，则图没有环。