# 排序算法

通常来说，排序的目的是快速查找。

衡量排序算法的优劣：

1.时间复杂度：分析关键字的比较次数和记录的移动次数

2.空间复杂度：分析排序算法中需要多少辅助内存

3.稳定性：若两个记录A和B的关键字值相等，但排序后A、B的先后次序保持不变，则称这种排序算法是稳定的。

排序算法分类：内部排序和外部排序。

内部排序：整个排序过程不需要借助于外部存储器（如磁盘等），所有排序操作都在内存中完成。

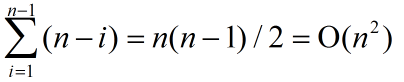
外部排序：参与排序的数据非常多，数据量非常大，计算机无法把整个排序过程放在内存中完成，必须借助于外部存储器（如磁盘）。外部排序最常见的是多路归并排序。可以认为外部排序是由多次内部排序组成。

## 选择排序

将待排序的元素分为已排序（初始为空）和未排序两组，依次将未排序的元素中值最小的元素放入已排序的组中。

### 直接选择排序

直接选择排序简单直观，但性能略差

算法的时间效率：无论初始状态如何，在第i趟排序中选择最小关键码的元素，需做n-i次比较，因此总的比较次数为：

算法的空间效率：空间效率很高，只需要一个附加程序单元用于交换，其空间效率为

算法的稳定性：不稳定

### 堆排序

堆排序是一种较为高效的选择排序方法，但实现起来略微复杂

算法的时间效率：假设有n个数据，需要进行n-1次建堆，每次建堆本身耗时 ，则其时间效率为

算法的空间效率：空间效率很高，只需要一个附加程序单元用于交换，其空间效率为

算法的稳定性：不稳定

## 交换排序

相邻两元素进行比较，如有需要则进行交换，每完成一次循环就将最大元素排在最后（如从小到大排序），下一次循环是将其它的数进行类似操作。

### 冒泡排序

算法的时间效率：从冒泡排序的算法可以看出，若待排序的元素为正序，则只需进行一趟排序，比较次数为n-1次，移动元素次数为0；若待排序的元素为逆序，则需要进行n-1趟排序，比较次数为，移动次数为，因此时间复杂度为

算法的空间效率：空间效率很高，只需要一个附加程序单元用于交换，其空间效率为

算法的稳定性：稳定

### 快速排序

算法的时间效率：时间效率很好，因为每趟能确定的元素都呈指数增长，故时间复杂度为

算法的空间效率：由于使用递归，而递归使用栈，因此空间效率最优时为

算法的稳定性：由于包含跳跃式交换，因此不稳定

## 插入排序

每次将一个待排序的元素，按其关键字的大小插入到前面已经排好序的子文件的适当位置，直到全部记录插入完成为止。

### 直接插入排序

直接插入排序的基本思想是：把n个待排序的元素看成为一个有序表和一个无序表，开始时有序表中只包含一个元素，无序表中包含有n-1个元素，排序过程中每次从无序表中取出第一个元素，把它的排序码一次与有序表元素的排序码进行比较，将它插入到有序表中的适当位置，使之成为新的有序表。

算法的时间效率：在最坏的情况下，所有元素的比较次数总和为（0+1+2+…+n-1）= ；在其他情况下，也要考虑移动元素的次数，故时间复杂度为

算法的空间效率：空间效率很高，只需要一个附加程序单元用于交换，其空间效率为

算法的稳定性：稳定

### 折半插入排序

折半插入排序是对直接插入排序的简单改进。

此处介绍的折半插入，其实就是通过不断地折半来快速确定第i个元素的插入位置，这实际上是一种查找算法：折半查找。Java的Arrays类里的binarySearch()方法，就是折半查找的实现，用于从指定数组中查找指定元素，前提是该数组已经处于有序状态。

与直接插入排序的效果相同，只是更快了一些，因为折半插入排序可以更快地确定第i个元素的插入位置

### Shell排序

先将整个待排元素序列分割成若干个子序列（由相隔某个“增量”的元素组成）分别进行直接插入排序，待整个序列中的元素基本有序（增量足够小）时，再对全体元素进行一次直接插入排序。因为直接插入排序在元素基本有序的情况下（接近最好情况），效率是很高的，因此希尔排序的时间效率上有较大提高。

算法的时间效率：开销估计在 之间

算法的空间效率：空间效率很高，只需要一个附加程序单元用于交换，其空间效率为

算法的稳定性：不稳定

## 归并排序

二路归并排序的基本思想是将两个有序表合并为一个有序表。顺序比较两者的相应元素，小者移入到第三个表中，如此反复，直至其中任一表都移入第三个表为止。

算法的时间效率：归并算法需要递归地进行分解、合并，每进行一趟归并排序，需要merge()方法一次，每次执行merge()需要比较n次，故复杂度为

算法的空间效率：较差，需要一个与原始序列同样大小的辅助序列

算法的稳定性：稳定

## 桶式排序

桶式排序不再是一种基于比较的排序方法，它是一种非常巧妙的排序方式，但这种排序方式需要待排序列满足如下两个特征：

待排序列的所有值处于一个可枚举范围内

待排序列所在的这个可枚举范围不应该太大，否则排序开销太大

算法的时间效率：时间效率极高，只需经过两轮遍历即可

算法的空间效率：空间开销较大，需要两个数组来完成

算法的稳定性：稳定

## 基数排序

基数排序已经不再是一种常规的排序方法，它更多地像是一种排序方法的应用，基数排序必须依赖于另外的排序方法。基数排序的总体思路就是将待排数据拆分成多个关键字进行排序，也就是说，基数排序的实质是多关键字排序。

多关键字排序的思路是将待排数据里的排序关键字拆分成多个排序关键字：第1个子关键字、第2个子关键字、第3个子关键字……然后，根据子关键字对待排数据进行排序。

在进行多关键字排序时有两种解决方案：

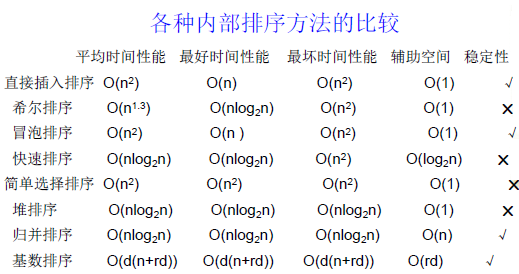
最高位优先法：MSD

最低位优先法：LSD

比较MSD法和LSD法，一般来讲，LSD法要比MSD法来得简单，因为LSD法是从头到尾进行若干次分配和收集，执行的次数取决于构成关键字值的成分为多少；而MSD法则要处理各序列与子序列的独立排序问题，就可能复杂一些。

## 不同排序算法的比较和选用

### 排序方法的比较



1.从平均时间而言：快速排序最佳。但在最坏情况下时间性能不如堆排序和归并排序。

2.从算法简单性看：由于直接选择排序、直接插入排序和冒泡排序的算法比较简单，将其认为是简单算法，都包含在上图的“简单排序”中。对于Shell排序、堆排序、快速排序和归并排序算法，其算法比较复杂，认为是复杂排序。

3.从稳定性看：直接插入排序、冒泡排序和归并排序时稳定的；而直接选择排序、快速排序、 Shell排序和堆排序是不稳定排序

4.从待排序的记录数n的大小看，n较小时，宜采用简单排序；而n较大时宜采用改进排序。

### 排序方法的选择

(1)若n较小(如n≤50)，可采用直接插入或直接选择排序。当记录规模较小时，直接插入排序较好；否则因为直接选择移动的记录数少于直接插入，应选直接选择排序为宜。

(2)若文件初始状态基本有序(指正序)，则应选用直接插入、冒泡或随机的快速排序为宜；

1. 若n较大，则应采用时间复杂度为O(nlgn)的排序方法：快速排序、堆排序或归并排序。

# 限流算法

**服务器容量**(服务器能够允许最大连接数)

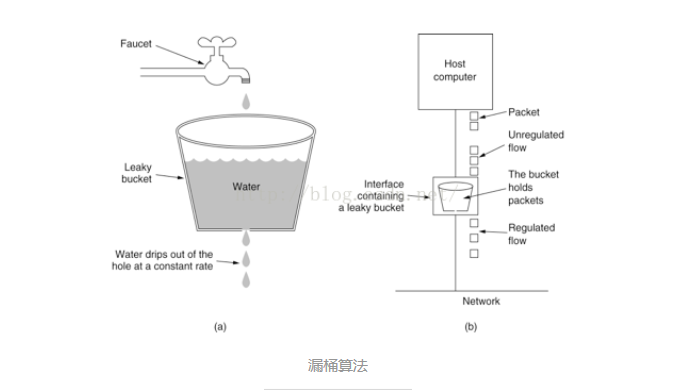
**服务器流量**(服务器的吞吐能力，一般是指服务器在单位时间内能够处理的请求的数量)

按秒，计算服务器每秒的请求数量是否达到最大值(10000个请求)，如果超过该值，则拒绝请求

第一秒中的第999毫秒，和第二秒中的第1毫秒之间

## 漏桶算法

漏桶算法很好的解决了时间边界处理不够**平滑**的问题，在每次请求进桶前都将执行“漏水”的操作，然后再计算当前水量，即不以时间为界限，而以流量为界限进行计算，回避了**时间边界**的问题。

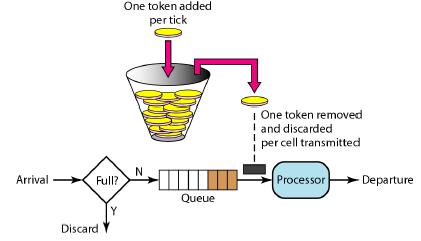


**伪代码**

|  |
| --- |
| long timeStamp = getNowTime(); int capacity = 10000;// 桶的容量，即最大承载值 int rate = 1;//水漏出的速度，即服务器的处理请求的能力 int water = 100;//当前水量，即当前的即时请求压力  //当前请求线程进入漏桶方法，true则不被拒绝，false则说明当前服务器负载水量不足，则被拒绝 public static bool control() {  long now = getNowTime();//当前请求时间  //先执行漏水代码  //rate是固定的代表服务器的处理能力，所以可以认为“时间间隔\*rate”即为漏出的水量  water = Math.*max*(0, water - (now - timeStamp) \* rate);//请求时间-上次请求时间=时间间隔  timeStamp = now;//更新时间，为下次请求计算间隔做准备  if (water < capacity) {  // 执行漏水代码后，发现漏桶未满，则可以继续加水，即没有到服务器可以承担的上线  water ++;  return true;  } else {  return false;//水满，拒绝加水，到服务器可以承担的上线，拒绝请求  } } |

## 令牌桶算法

令牌桶算法的原理是系统会以一个恒定的速度往桶里放入令牌，而如果请求需要被处理，则需要先从桶里获取一个令牌，当桶里没有令牌可取时，则拒绝服务。



**伪代码**

|  |
| --- |
| long timeStamp=getNowTime(); int capacity; // 桶的容量  int rate ;//令牌放入速度 int tokens;//当前水量   bool control() {  //先执行添加令牌的操作  long now = getNowTime();  tokens = max(capacity, tokens+ (now - timeStamp)\*rate);  timeStamp = now;  if(tokens<1){  return false; //令牌已用完，拒绝访问  }else{  tokens--;  retun true; //还有令牌，领取令牌  } } |

# 加密算法

## DES加密

## RSA加密