# 经典算法讲解

## 排序

### 1 快排

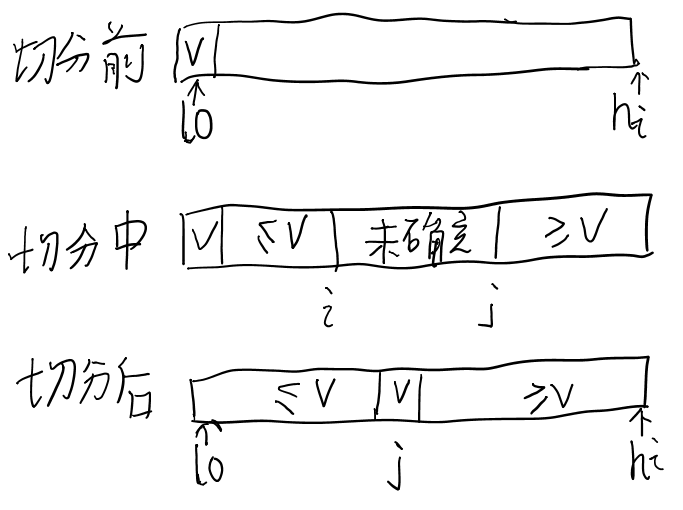
**主要思想**：

快排的思想是当两个子数组都有序时，整个数组也就自然有序了。所以，每次将数组一分为二，并令右边子数组的元素大于左边子数组的元素。

快排的关键就在于切分(partition)，这个过程使得数组满足下面三个条件：

* 对于某个j，a[j]已经排定
* a[lo]到a[j-1]中的所有元素都不大于a[j];
* a[j+1]到a[hi]中的所有元素都不小于a[j]

一般策略是取a[lo]为切分元素，从数组左端开始向右扫描直到找到一个大于等于它的元素，再从数组右端开始向左扫描直到找到一个小于等于它的元素，交换他们的位置。示意图如下：



快排还有一个特点，它是原地切分的，这很节省空间。

**代码实现**：

|  |
| --- |
| **public class** QuickSort {  **public static void** sort(Comparable[] a) {  sort(a, 0, a.**length**-1);  }   /\*\*  \* 快排  \*/  **private static void** sort(Comparable[] a, **int** lo, **int** hi) {  **if**(lo >= hi) **return**;  **int** part = partition(a, lo, hi);  sort(a, lo, part-1);  sort(a, part+1, hi);  }   /\*\*  \* 分区操作  \* **@param a** 要分区的数组  \* **@param lo** 要分区的左边界索引  \* **@param hi** 要分区的右边界索引  \* **@return** pivot 一个中心点，  \* 保证当i<pivot,Comparable[i] <= Comparable[pivot]；当i>pivot,Comparable[i]>=Comparable[pivot]  \*/  **private static int** partition(Comparable[] a, **int** lo, **int** hi) {  **int** i = lo, j = hi+1;  **while**(**true**) {  **while**(less(a[++i], a[lo])) {  **if**(i == hi)  **break**;  }  **while**(less(a[lo], a[--j]));  **if**(i >= j) **break**;  exch(a, i, j);  }  //将a[lo]与a[j]交换，为什么是j呢，因为a[j]<=a[lo],a[i]>=a[lo]  exch(a, lo, j);  **return** j;  }   **private static boolean** less(Comparable i, Comparable j) {  **return** i.compareTo(j) < 0;  }   **private static void** exch(Comparable[] a, **int** i, **int** j) {  Comparable temp = a[j];  a[j] = a[i];  a[i] = temp;  } } |

**算法改进**：

1. 切换到插入排序

对于小数组，快速排序比插入排序慢，因为递归，快速排序的sort()方法在小数组中也会调用自己。

只需将代码中标亮的内容换为：

|  |
| --- |
| if(hi <= lo + M) {Insertion.sort(a, lo, hi); return;} |

M通常取5~15之间的任意值

1. 三取样本法

即取三个数中的中位数为pivot,这样切分更好

1. 熵最优的排序

将数组切分为三部分，分别对应小于、等于和大于切分元素的数组元素

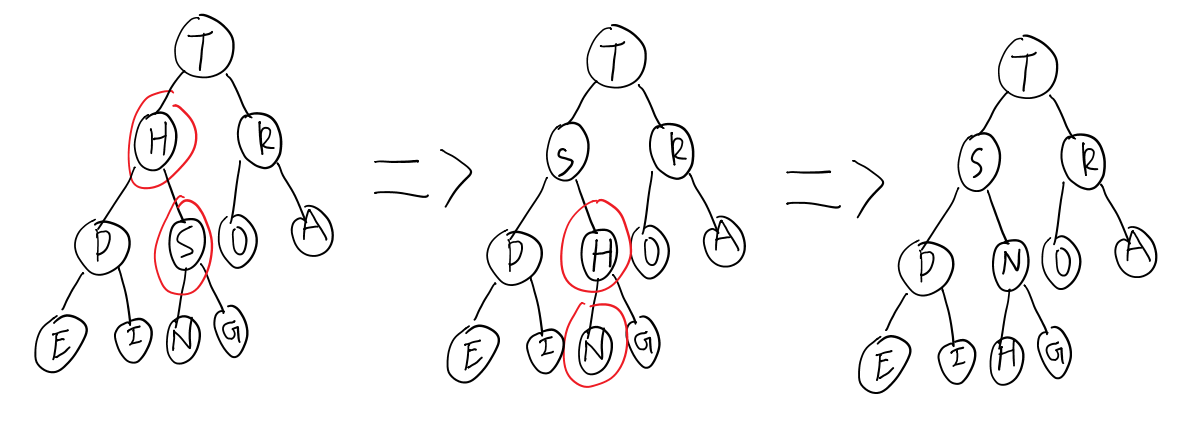
### 2 堆排序

**主要思想**：

要讲堆排序，我们首先要了解一下堆：堆分为大根堆和小根堆，其中大根堆保证每个节点都大于它的子节点，但子节点之间没有关系；小根堆保证每个节点都小于它的子节点。

堆排序的思想是我们将所有待排序的元素构建成最大完全二叉堆，因为完全二叉堆可以用数组表示，所以我们就可以每次将堆的最大根与数组最后一个元素交换，然后重新建堆，依次循环，直到交换N-2次。

堆排序的关键操作就是建堆，在讲建堆之前，我们先讲一下堆有序化过程。堆有序化过程就是通过堆中节点下沉(sink)或上浮(swim)来保持堆有序的过程，这里以下沉为例：



接下来讲建堆，一种方法是我们从左向右遍历数组，然后对扫描指针元素执行上浮操作(swim)，这样就能保证扫描指针元素左侧是一个有序的完全二叉堆，但是这样的时间复杂度是O(NlogN)。更高效的方法是从右向左遍历数组，然后对扫描指针元素执行下沉操作(sink)，可证明这是O(N)的时间复杂度。

最后一步就是，每次将根元素与未排序数组的尾部元素交换，并执行以下下沉操作。

值得注意的是，这里堆节点的索引从0开始，若一个节点的索引为i，那么它的左子节点的索引为2\*i+1，它的右子节点的索引为2\*i+2。

**代码实现**：

|  |
| --- |
| **public class** HeapSort {  **public static void** sort(**int**[] nums) {  **int** N = nums.**length**;  //建堆操作，大根堆，堆索引是从0开始的  **for**(**int** i=N/2; i>=0; i--) {  sink(nums, i, N);  }  //堆排序，堆索引是从0开始的  **while**(N > 1) {  exch(nums, 0, --N);  sink(nums, 0, N);  }  }   /\*\*  \* 下沉操作，如果当前节点小于它子节点，就交换它和它的子节点  \* **@param nums** 存放堆的数组  \* **@param i** 下沉节点的开始索引  \* **@param N** 下沉操作的尾边界，不包含N  \*/  **private static void** sink(**int**[] nums, **int** i, **int** N) {  //节点索引是从0开始的，i\*2+1代表它的左子节点的索引，保证小于尾边界N  // 0  // / \  // 1 2  **while**(i\*2+1 < N) {  **int** k = i\*2+1;  //如果右子节点存在，且大于左子节点  **if**(k < N-1 && nums[k+1] > nums[k]) k++;  **if**(nums[k] < nums[i]) **break**;  exch(nums, i, k);  i = k;  }  }   **private static void** exch(**int**[] nums, **int** i, **int** j) {  **int** temp = nums[j];  nums[j] = nums[i];  nums[i] = temp;  } } |

## 算法比较

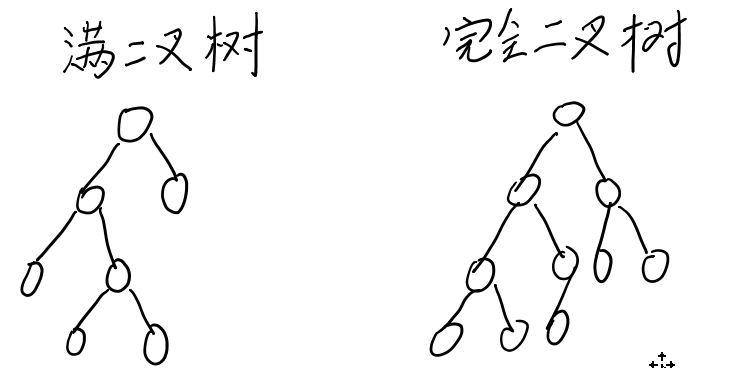
## 字符串

### 1 KMP

## 树

一些基本常识：

满二叉树 vs 完全二叉树



### 1 遍历方式

**基本概念**：

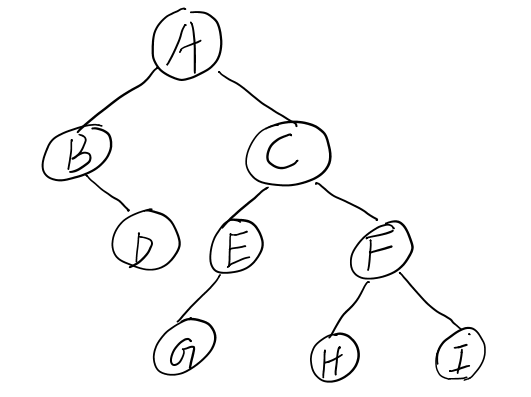
按照遍历根的顺序将遍历方式分为前序、中序和后序遍历三种方式，方式如下：

前序：根左右

中序：左根右

后序：左右根

对于下面这棵树，三种遍历方式的结果如下：



前序：A B D C E G F H I

中序：B D A G E C H F I

后序：D B G E H I F C A

**代码实现**：

* 1. 前序

递归方式：

|  |
| --- |
|  |

* 1. 中序
  2. 后序

**补充**：

给定一棵树的前序遍历和中序遍历，或者是中序遍历和后序遍历，能确定唯一一棵树，而给定一个前序和一个后序遍历，则不能唯一确定树。

## 图

### 1 遍历方式