面试宝典-数据结构与算法

* 数据结构

线性表

堆

set 集合

树

常见编程题

如何判断链表中是否有环

- 方法一：暴力双重循环
- 方法二：使用 HashSet

  在方法一的基础上进行优化降低复杂度，使用hashSet作为额外缓存，可以减少一层循环，具体思路如下:
  首先创建一个以节点ID为Key的HashSet集合，用来存储曾经遍历过的节点。然后同样从头节点开始，依次遍历单链表中的每一个节点。每遍历一个新节点，都用新节点和HashSet集合中存储的节点进行比较，如果发现HashSet中存在与之相同的节点ID，则说明链表有环，如果HashSet中不存在与新节点相同的节点ID，就把这个新节点ID存入HashSet中，之后进入下一节点，继续重复刚才的操作。
  使用HashSet将算法的时间复杂度降为了O（n）.
- 方法三：利用两个指针

  首先创建两个指针p1和p2（在Java里就是两个对象引用），让它们同时指向这个链表的头节点。然后开始一个大循环，在循环体中，让指针p1每次向后移动1个节点，让指针p2每次向后移动2个节点，然后比较两个指针指向的节点是否相同。如果相同，则可以判断出链表有环，如果不同，则继续下一次循环。

```
public static boolean isLinkCycle(Node head){
  Node p1 = head;
  Node p2 = head;
  while(p2 != null && p2.next != null){
    p1 = p1.next;
    p2 = p2.next.next;
    if(p1 == p2){
      return true;
    }
    return false;
}
```

判断链表是否有环以及找入口

双指针：快慢指针

变量：首先定义 fast 和 slow 快慢指针。

操作：从头结点出发，fast指针每次移动两个结点，slow指针每次移动一个结点。如果fast和slow指针在途中相遇，说明这个链表有环返回true。如果fast最后走到了空结点，说明这个链表没有环返回false。

为什么fast一次走两个结点，slow一次走一个结点，如果有环的话，一定会在环内相遇呢，而不会永远的错开?

首先明确一点：如果有环，fast指针一定是先入环，slow指针后入环，如果要相遇那么一定会在环内相遇。fast指针一定会在slow指针的前面。
其次：为什么fast指针和slow指针一定会相遇呢?

我们知道 fast 指针一次移动两个结点，slow指针一次移动一个结点。fast 指针相对于 slow 指针每次移动一个结点，然后又是在环内，fast 又在 slow 的 前 面，就相当于fast在slow后面追赶slow,每次向slow靠近一步。所以最后一定就会把slow指针追上。

slow指针在环内走过的结点会不会超过一圈?

设环的长度为 C 时，考虑极限最大情况下fast和slow指针相差C-1个结点。当在slow指针进入环后，设fast指针走了Y的长度，slow指针走了X的长度，所以有等式 Y-X=C-1, Y=2*X, ==> X = C-1。所以在极限情况下慢指针进入环内最多在环中走了C-1个结点，所以不会超过一圈。

- fast 指针在环内走过的结点最少是多少呢?

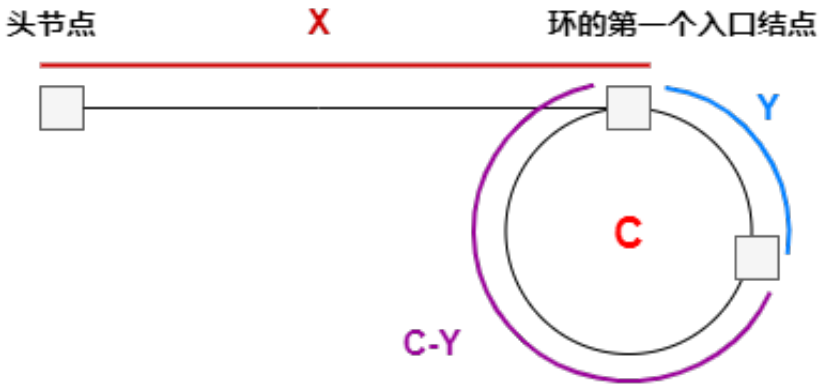
设环的长度为 C 时，考虑极限最小情况下fast和slow指针相差0个结点，即慢指针刚刚进入环就遇见快指针。当在slow指针进入环后，设fast指针走了Y的长度，slow指针走了X的长度，所以有等式 Y-X=0, Y=2*X, ==> Y= 0。但是之前快指针最少已经走了C个节点，因此快指针进入环后相遇慢指针最少走了C个节点。

fast 指针为什么要一次走两个节点，一次走三步节点，一次走四个节点，最终还会不会和slow指针相遇呢?

答案是：可能会相遇，当快指针走三步时，快指针就相对于慢指针走2步，当最后慢指针和快指针刚刚相差一个节点时，这时快指针就刚刚跨过慢指针，所以这次就不会相遇了。走四步也是一样的。所以当快指针走两步时，就一定会相遇，因为快指针只相对于慢指针走一步，每次靠近一步，所以最后一定会相遇慢指针。

找到环之后怎么样找第一个入口，以及怎么证明。

当我们能够判断链表有环时，这下我们的目的就是怎样找到环的入口结点了。



变量：假设头节点到环的第一个入口节点的节点数为X。环形入口结点到fast指针和slow指针相遇节点 节点数为Y。环的节点数为C。那么相遇节点到环第一个入口节点的节点数为 C-Y。

证明：
■相遇时慢指针走过的节点数为X+Y：我在上面已经给出了证明慢指针在环内走过的节点数不会超过一圈。所以慢指针在环中走过的长度就是Y.
■相遇时快指针走过的节点数为X+NC+Y(N>=1)：我们不知道环的大小，当环很小时，快指针可能在环内已经走了很多圈了。
■相遇时有等式 X+NC+Y = 2(X+Y)：因为快指针是慢指针速度的两倍。解得等式有 X = NC-Y==>X = (N-1)C+C-Y。
■最后推出来的等式说明：从头节点出发一个指针，从相遇节点出发一个指针，这两个指针每次走一步，那么当这两个指针相遇时**就是环形入口的节点。因为当N=1时，等式刚刚是X = C-Y，所以刚好相遇时就是环形入口节点。当N>1时，也能找到相遇节点，如果N>1就是一个指针在环内多绕 N-1 圈最后还是找到了环形的入口节点。**

算法

递归

程序调用自身的编程技巧称为递归（recursion）。

在计算机科学领域中，递归是通过函数调用本身来实现的。每次成功调用都使得问题的答案范围越来越小，越来越接近问题的答案。

递归的优点与不足

递归的优点，在多数情况下，递归的实现更加优雅，代码逻辑的可读性更强。但是，如果递归的次数达到一定的数量就会抛出栈溢出错误。

如果使用循环，程序的性能可能更高；如果使用递归，程序可能更容易理解。

分治算法

分治法的设计思想是：将一个难以直接解决的大问题，分割成一些规模较小的相同问题，以便各个击破，分而治之。

分治策略是：对于一个规模为 n 的问题，若该问题可以容易的解决（比如说规模 n 较小）则直接解决，否则将其分解为 k 个规模较小的子问题，这些子问题互相独立且与原问题相似相同，递归地解这些子问题，然后将各子问题的解合并得到原问题的解。这种算法设计策略叫做分治法。

分治法的基本步骤

分治法在每一次递归上都有三个步骤：

1. step 1 分解：将原问题分解为若干个规模较小，相互独立，与原问题形式相同的子问题；
2. step 2 解决：若子问题规模较小而容易被解决则直接解，否则递归地解各个子问题；
3. step 3 合并：将各个子问题的解合并为原问题的解。

分治法的复杂性分析

一个分治法将规模为 n 的问题分成 k 个规模为 n/m 的子问题去解。设分解阈值 n0 = 1，且 adhoc 解规模为 1 的问题耗费 1 个单位时间。再设将原问题分解为 k 个子问题以及用 merge 将 k 个子问题的解合并为原问题的解需用 f(n) 个单位时间。用 T(n) 表示分治法解规模为 |P|=n 的问题所需的计算时间，则有：

T(n) = k T(n/m) +f(n)

通过迭代法求得方程的解。

递归方程及其解只给出 n 等于 m 的方幂时 T(n) 的值，但是如果认为 T(n) 足够平滑，那么由 n 等于 m 的方幂时 T(n) 的值可以估计T(n) 的增长速度。通常假定 T(n) 是单调上升的，从而当 mi<=n<mi+1 时，T(mi) <= T(n) < T(mi+1)。

可使用分治法求解的一些经典问题

1. 二分搜索
2. 大整数乘法
3. Strassen 矩阵乘法
4. 棋盘覆盖
5. 合并排序
6. 快速排序
7. 线性时间选择
8. 最接近点对问题
9. 循环赛日程表
10. 汉诺塔

分支定界法

类似于回溯法，也是一种在问题的解空间树 T 上搜索问题解的算法，但在一般情况下，分支限界法与回溯法的求解目标不同。回溯法的求解目标是找出 T 中满足约束条件的所有解，而分支限界法的求解目标则是找出满足约束条件的一个解，或是在满足约束条件的解中找出使某一目标函数值达到极大或极小的解，即在某种意义下的最优解。

回溯法与分支限定法的一些区别

有一些问题其实无论用回溯法还是分支限界法都可以得到很好的解决，但是另外一些则不然。也许需要具体一些的分析 -- 到底何时使用分支限界而何时使用回溯呢?

回溯法和分支限界法的一些区别：

- 方法对解空间树的搜索方式
- 存储结点的常用数据结构
- 结点存储特性常用应用

回溯法深度优先搜索堆栈活结点的所有可行子节点，被遍历后才被从栈中弹出找出满足约束条件的所有解。

分支限界法广度优先或最小消耗优先搜索队列、优先队列每个结点只有一次成为活结点的机会，找出满足约束条件的一个解或特定意义下的最优解。

贪心算法

所谓贪心算法是指，在对问题求解时，总是做出在当前看来是最好的选择。也就是说，不从整体最优上加以考虑，它所做出的仅是在某种意义上的局部最优解。

必须注意的是，贪心算法不是对所有问题都能得到整体最优解，选择的贪心策略必须具备无后效性，即某个状态以后的过程不会影响以前的状态，只与当前状态有关。

所以对所采用的贪心策略一定要仔细分析其是否满足无后效性。

贪心算法适用的问题

贪心策略适用的前提是：局部最优策略能导致产生全局最优解。

实际上，贪心算法适用的情况很少。一般，对一个问题分析是否适用于贪心算法，可以先选择该问题下的几个实际数据进行分析，就可做出判断。

贪心策略的选择

因为用贪心算法只能通过解局部最优解的策略来达到全局最优解，因此，一定要注意判断问题是否适合采用贪心策略，找到的解是否一定是问题的最优解。

动态规划

动态规划（dynamic programming,DP）：将一个问题拆成几个子问题，分别求解这些子问题，即可推断出大问题的解。

DP 的核心思想：尽量缩小可能解空间。

查找算法

顺序查找

顺序查找的基本思想

顺序查找也称为线性查找，属于无序查找算法。从数据结构线性表的一端开始，顺序扫描，依次将扫描到的节点关键字和给定关键字 k 相比较，若相等则表示查找成功，若扫描结束仍没有找到关键字等于 k 的节点，表示查找失败。

顺序查找适合于存储结构为顺序存储或链式存储的线性表。

顺序查找的复杂度

查找成功时平均查找长度为：（假设每个元素的概率是相等的） ASL = 1/n(1+2+3+...+n) = (n+1)/2。

当查找不成功时，需要 n+1 比较，时间复杂度为 O(n)。

所以，顺序查找的时间复杂度为 O(n)。

二分查找

基本思想

二分查找（binary search），也称折半查找，是一种在有序数组中查找某一特定元素的搜索算法，属于有序查找算法。搜索过程从数组的中间元素开始，如果中间元素正好是要查找的元素，则搜索过程结束；如果某一特定元素大于或者小于中间元素，则在数组大小或小于中间元素的那一半中查找，而且跟开始一样从中间元素开始比较。如果在某一步骤数组为空，则代表找不到，这种搜索算法每一次比较都使搜索范围缩小一半。

元素必须是有序的，如果是无序的则要先进行排序操作。

注：折半查找的前提条件是需要有序表顺序存储，对于静态查找表，一次排序后不再变化，折半查找能得到不错的效率。但对于需要频繁执行插入或删除操作的数据集来说，维护有序的排序会带来不小的工作量，那就不建议使用。

- 时间复杂度：这种搜索每次都把搜索区域减少一半，时间复杂度为 O(logN)(N 代表集合中元素的个数)。
- 空间复杂度：O(1)。虽以递归形式定义，但是尾递归，可改写尾循环。

```
/**
 * 二分查找
 */
public class BinarySearch {

    /**
     * 递归
     *
     * @param array
     * @param low
     * @param high
     * @param key
     * @return
     */
    public static int binarySearch(int array[], int low, int high, int key) {
```

```
        if (low > high) return -1;
        int mid = low + (high - low) / 2;
        if (array[mid] > key) {
            return binarySearch(array, low, mid - 1, key);
        } else if (array[mid] < key) {
            return binarySearch(array, mid + 1, high, key);
        } else {
            return mid;
        }
    }

    /**
     * 非递归
     *
     * @param a
     * @param key
     * @return
     */
    public static int binarySearchWithoutRecusion(int a[], int key) {
        int low = 0;
        int high = a.length - 1;
        while (low <= high) {
            int mid = low + (high - low) / 2;
            if (a[mid] > key) {
                high = mid - 1;
            } else if (a[mid] < key) {
                low = mid + 1;
            } else {
                return mid;
            }
        }
        return -1;
    }

    public static void main(String[] args) {
        int a[] = new int[]{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
        System.out.println("search 7:" + binarySearch(a, 0, a.length - 1, 7));
        System.out.println("search 5:" + binarySearchWithoutRecusion(a, 5));
    }
}
```

复杂度

最坏情况下，关键字比较次数为 log2(n+1)，且期望时间复杂度为 O(log2n)。

二分查找中值的计算

如何计算二分查找中的中值，有两种计算方法：

1. 算法一：mid = (low+high)/2
2. 算法二：mid = low+(hight-low)/2

看起来是算法一简洁，算法二提取之后，跟算法一没有什么区别，但是实际上，区别是存在的。算法一的做法，在极端情况下，(low+high) 存在着溢出的风险，进而得到错误的 mid 结果，导致程序错误。而算法二能够保证计算出来的 mid，一定大于 low，小于 high，不存在溢出的问题。

二分查找的缺陷

二分查找法的 O(logN) 让它成为十分高效的算法。不过它的缺陷却也是那么明显的。就在它的限定之上：必须有序，很难保证数组都是有序的。当然可以在构建数组的时候进行排序，可是又落到了第二个瓶颈上：它必须是数组。

数组读取效率是 O(1)，可是它的插入和删除某个元素的效率却是 O(n)，因此导致构建有序数组变成低效的事情。

解决这些缺陷问题更好的方法应该是使用二叉查找树，最好自然是自平衡二叉查找树了，即能高效的（O(NlogN)）构建有序元素集合，又能如同二分查找法一样快速（O(logN)）的搜寻目标数。

插值查找

插值查找的基本思想

二分查找是折半查找，而不是折四分之一或者折更多。

在字典里面查 “ apple ”，肯定是有目的的往前翻，而查找 “ zoo ”，那肯定是往后翻，绝对不会从中间开始查起。

同样的，比如要在取值范围 1~10000 之间 100 个元素从小到大均匀分布的数组中查找 5，自然会考虑从数组下标较小的开始查找。

经过以上分析，折半查找这种查找方式，不是自适应的。二分查找中查找点计算如下：

mid = (low+high)/2，即 mid = low + 1/2*(high-low)。

通过类比，可以将查找的点改进为如下：

mid = low + (key-a[low])/(a[high]-a[low])*(high-low)。

也就是将上述的比例参数 1/2 改进为自适应的，根据关键字在整个有序表中所处的位置，让 mid 值的变化更靠近关键字 key，这样也就间接地减少了比较次数。

基本思想：基于二分查找算法，将查找点的选择改进为自适应选择，可以提高查找效率。当然，插值查找也属于有序查找。

注：对于表长较大，而关键字分布又比较均匀的查找表来说，插值查找算法的平均性能比折半查找要好的多。反之，数组中如果分布非常不均匀，那么插值查找未必是很合适的选择。

插值查找的时间复杂度

查找成功或者失败的时间复杂度均为 O(log2(log2n))。

插值查找的代码

与二分查找基本一样，就 mid 的计算方式不一样。

mid = begin + (key-a[begin])/(a[end]-a[begin])*(end-begin)。

```
/**
 * 插值查找
 */
public class InsertionSearch {
    /**
     * 用插值查找查找在 nums 数组中查找 key 的 index
     * 先用快排对数组进行排序，然后设定 begin = 0,end = length-1
     * mid = begin+(key-nums[begin])/(nums[end]-nums[begin])*(end-begin)，查找
mid 的值与 key 的大小
     * 如果相同，返回 index
     * 如果 mid < key，那么 begin = mid+1，如果 mid > key，那么 end = mid-1
     * 然后循环，直到 end < begin，返回 -1
     *
     * @param nums
     * @param key
     * @return 返回 key 在 nums 数组中的下标，没有数组中没有这个 key，返回 -1
     */
    public static int insertionSearch(int[] nums, int key) {
        int length = nums.length;
        Arrays.sort(nums);
        // begin  = 0,end = length-1
        int begin = 0;
        int end = length - 1;
        // 循环，直到 end<begin，返回 -1
        while (begin <= end) {
            int mid = begin + (key - nums[begin]) / (nums[end] - nums[begin]) *
(end - begin);
            int now = nums[mid];
            if (now == key) {
                // 如果相同，返回 index
                return mid;
            }
            if (now < key) {
                // 如果 mid < key，那么 begin = mid+1
                begin = mid + 1;
            }
            if (now > key) {
                // 如果 mid > key，那么 end = mid-1
                end = mid - 1;
            }
```

```
            }
            return -1;
        }
    }
```

斐波那契查找

二叉树查找

二叉树查找的基本思想

二叉查找树是先对待查找的数据进行生成树，确保树的左分支的值小于右分支的值，然后再进行和每个结点的父节点比较大小，查找最适合的范围。这个算法的查找效率很高，但是如果使用这种查找方法要首先创建树。

二叉查找树（BinarySearch Tree，也叫二叉搜索树，或称二叉排序树 Binary Sort Tree）或者是一棵空树，或者是具有下列性质的二叉树：

1. 若任意节点的左子树不空，则左子树上所有结点的值均小于它的根结点的值。
2. 若任意节点的右子树不空，则右子树上所有节点的值均大于它的根节点的值。
3. 任意节点的左、右子树叶分别为二叉查找树。

二叉查找树性质：对二叉查找树进行中序遍历，即可得到有序的数列。

根据二叉查找树的性质，可以从根根据大小比较，然后到左右孩子搜索到对应的节点。

二叉树查找的复杂度

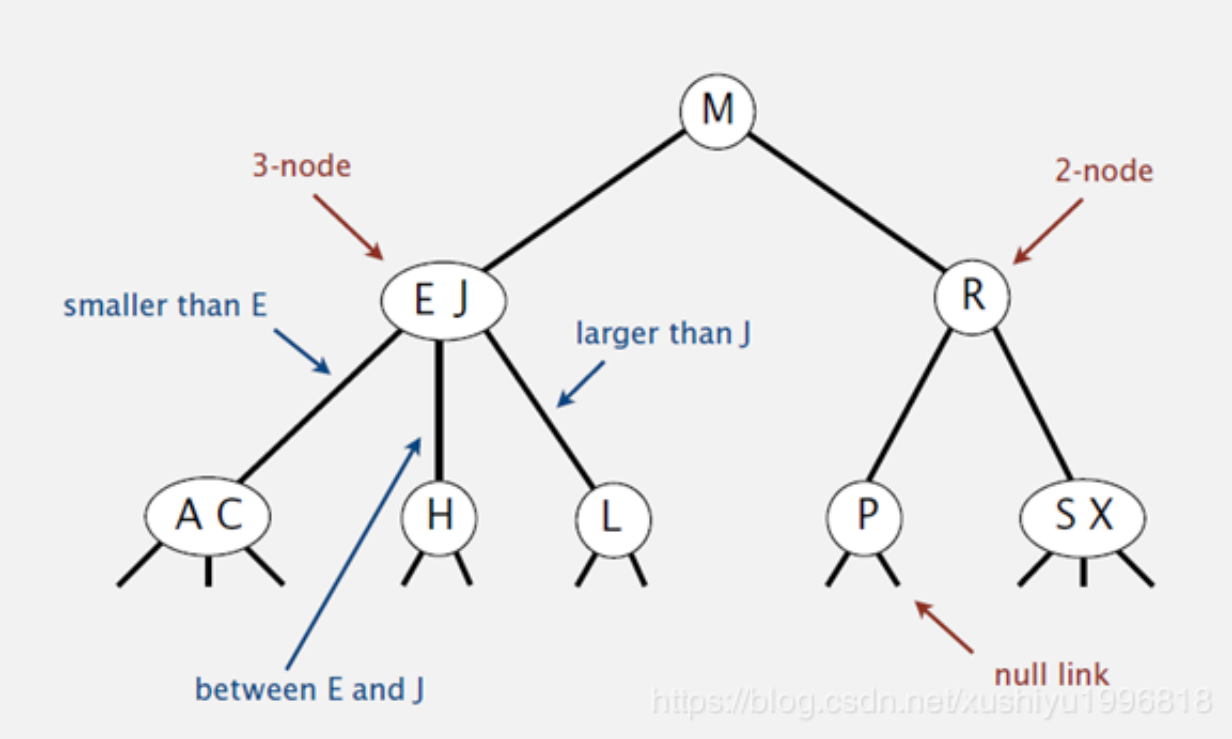
它和二分查找一样，插入和查找的时间复杂度均为 O(log2 n)，但是在最坏的情况下仍然会有 O(n) 的时间复杂度。原因在于插入和删除元素的时候，树没有保持平衡。为了追求的是在最坏的情况下仍然有较好的时间复杂度，引入了平衡查找树。

平衡查找数之2-3查找树

2-3 树的基本思想

2-3 查找树的定义：和二叉树不一样，2-3 树每个节点保存 1 个或者 2 个的值。对于普通的 2 节点（2-nodes），它保存 1 个 key 和左右 2 个子节点。对应 3 节点（3-nodes），保存两个 key，2-3 查找树的定义如下：

1. 要么为空，要么就是 2 节点或者 3 节点。
2. 对于 2 节点，该节点保存一个 key 及对应 value，以及 2 个指向左右节点的节点，左节点也是一个 2-3 节点，所有的值比 key 要小，右节点也是一个 2-3 节点，所有的值比 key 要大。
3. 对于 3 节点，该结点保存两个 key 及对应 value，以及三个指向左中右的节点。左节点也是一个 2-3 节点，所有的值均比两个 key 中的最小 key 还要小；中间节点也是一个 2-3 节点，中间节点的 key 值在两个根节点 key 值之间；右节点的所有 key 比两个 key 中的最大的 key 还要大。



2-3 查找树的性质

1. 如果中序遍历 2-3 查找树，就可以得到排好序的序列。
2. 在一个完全平衡的 2-3 查找树中，根节点到每一个为空节点的距离都相同。（这也是平衡树中“平衡”一词的概念，根节点到叶节点的最长距离对应于查找算法的最坏情况，而平衡树中根节点到叶结点的距离都一样，最坏情况也具有对数复杂度）。

2-3 树的复杂度

2-3 树的查找效率与树的高度是息息相关的。

在最坏的情况下，也就是所有的节点都是 2-node 节点，查找效率为 lgN。

在最好的情况下，所有的节点都是 3-node 节点，查找效率为 log3N 约等于 0.631lgN。

距离来说，对于一百万个节点的 2-3 树，树的高度为 12-20 之间，对于 10 亿个节点的 2-3 树，树的高度为 18-30 之间。

对于插入来说，只需要常数次操作即可完成，因为他只需要修改与该节点关联的节点即可，不需要检查其他节点，所以效率和查找类似。

平衡查找树之红黑树

红黑树的基本思想

2-3 查找树能保证在插入元素之后能保持树的平衡状态，最坏情况下即所有的子节点都是 2-nodes，树的高度为 lgn，从而保证了最坏情况下的时间复杂度。但是 2-3 树实现起来比较复杂，于是就有了一种简单实现 2-3 树的数据结构，即红黑树（Red-Black Tree）。

基本思想：红黑树的思想就是对 2-3 查找树进行编码，尤其是对 2-3 查找树中的 3-nodes 节点添加额外的信息。红黑树中将节点之间的链接分为两种不同类型，红色链接，他用来链接两个 2-nodes 节点来表示一个 3-nodes 节点。黑色链接用来链接普通的 2-3 节点。特别的，使用红色链接的两个 2-nodes 来表示一个 3-nodes 节点。特别的，使用红色链接的两个 2-nodes 来表示一个 3-nodes 节点，并且向

左倾斜，即一个 2-node 是另一个 2-node 的左子节点。这种做法的好处是查找的时候不用做任何修改，和普通的二叉查找树相同。

红黑树的定义

红黑树是一种具有红色和黑色链接的平衡查找树，同时满足：

- 红色节点向左倾斜
- 一个节点不可能有两个红色链接
- 整个树完全黑色平衡，即从根节点到叶子节点的路径上，黑色链接的个数都相同。

红黑树的性质：这个树完全黑色平衡，即从根节点到叶子节点的路径上，黑色链接的个数都相同（2-3 树的第 2 个性质，从根节点到叶子节点的距离都相等）。

复杂度分析：最坏的情况就是，红黑树中除了最左侧路径全部是由 3-node 节点组成，即红黑相间的路径长度是全黑路径长度的 2 倍。

B 树和 B+ 树

B 树和 B+ 树的基本思想

平衡查找树中的 2-3 树以及其实现红黑树。2-3 树中，一个节点最多有 2 个 key，而红黑树则使用染色的方式来标识这两个 key。

维基百科对 B 树的定义为 ： 在计算机科学中，B 树（B-tree）是一种树状数据结构，它能够存储数据、对其进行排序并允许以 O（log n）的时间复杂度运行进行查找、顺序读取、插入和删除的数据结构。B 树，概括来说是一个节点可以拥有多于 2 个子节点的二叉查找树。与自平衡二叉查找树不同，B 树为系统最优化大块数据的读和写操作。B-tree 算法减少定位记录时所经历的中间过程，从而加快存取速度。普遍应用于数据库和文件系统 。

B 树定义：

B 树可以看作是对 2-3 查找树的一种扩展，即它允许每个节点有 M-1 个子节点。

根节点至少有两个子节点。

每个节点有 M-1 个 key，并且以升序排列。

位于 M-1 和 M key 的子节点的值位于 M-1 和 M key 对应的 value 之间。

其他节点至少有 M/2 个子节点。

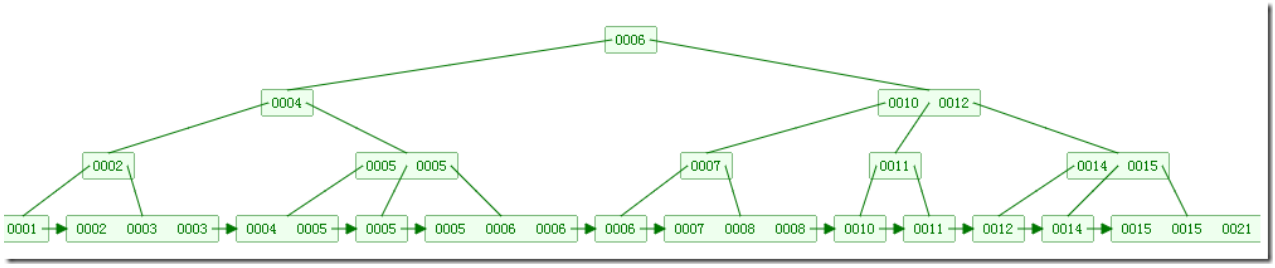
B 树是 2-3 树的一种扩展，它允许一个节点有多于 2 个的元素。B 树的插入及平衡化操作和 2-3 树很相似。

B+ 树定义：

B+ 树是对 B 树的一种变形树，它与 B 树的差异在于：

- 有 k 个子节点的节点必然有 k 个关键码；
- 非叶节点仅具有索引作用，跟记录有关的信息均存放在叶节点中。
- 树的所有叶结点构成一个有序链表，可以按照关键码排序的次序遍历全部记录。

B 和 B+ 树的区别在于，B+ 树的非叶子节点只包含导航信息，不包含实际的值，所有的叶子节点和相连的节点使用链表相连，便于区间查找和遍历。



B+ 树的优点在于：

由于 B+ 树在内部节点上不包含数据信息，因此在内存页中能够存放更多的 key。数据存放的更加紧密，具有更好的空间局部性。因此访问叶子节点上关联的数据也具有更好的缓存命中率。

B+ 树的叶子节点都是相连的，因此对整棵树的遍历只需要一次性遍历叶子节点即可。而且由于数据顺序排列并且相连，所以便于区间查找和搜索。而 B 树则需要进行每一层的递归遍历。相邻的元素可能在内存中不相邻，所以缓存命中性没有 B+ 树好。

但是 B 树也有优点，其优点在于，由于 B 树的每一个节点都包含 key 和 value，因此经常访问的元素可能离根节点更近，因此访问也更迅速。

B/B+ 树常用于文件系统和数据库系统中，它通过对每个结点存储个数的扩展，使得对连续的数据能够进行较快的定位和访问，能够有效减少查找时间，提高存储的空间局部性从而减少 IO 操作，它广泛用于文件系统及数据库中，如：

Windows：HPFS 文件系统；

Mac：HFS，HFS+ 文件系统；

Linux：ResiserFS、XFS、Ext3FS、JFS 文件系统；

数据库：ORACLE、MYSQL、SQLSERVER 等中。

B 树和 B+ 树的复杂度

在 n 个关键字的 m 阶 B 树和 B+ 查找，从根节点到关键字所在的节点所涉及的节点数不超过：

$$\log_{\lceil m/2 \rceil}(\frac{n+1}{2})+1$$

可以认为时间复杂度就是这个，因为 B 树和 B+ 树一般都是在文件系统中用，io 的速度比在内存中计算的速度慢很多。

总时间 = 查询节点次数 * io 单次时间 + 查询节点次数 * log2 阶数 * 内存中比较时间。

可以认为总时间约等于查询节点次数 * io 单次时间。

可以认为 B 树和 B+ 树，时间复杂度为 O（log m/2 n/2）。

树表查找总结

二叉查找树平均查找性能不错，为 O(logn)，但是最坏情况会退化为 O(n)。在二叉查找树的基础上进行优化，可以使用平衡查找树。平衡查找树中的 2-3 查找树，这种数据结构在插入之后能够进行自平衡操作，从而保证了树的高度在一定的范围内进而能够保证最坏情况下的时间复杂度。但是 2-3 查找树实现起来比较困难，红黑树是 2-3 树的一种简单高效的实现，他巧妙的使用颜色标记来替代 2-3 树种比较难处理的 3-node 节点问题。红黑树是一种比较高效的平衡查找树，应用非常广泛，很多编程语言的内部实现都或多或少的采用了红黑树。

除此之外，2-3查找树的另一个扩展 -- B/B+ 平衡树，在文件系统和数据库系统中有着广泛的应用。

分块查找

分块查找的基本思想

对于需要经常增加或减少数据的数据元素列表，每次增加或减少数据之后排序，或者每次查找前排序都不是很好的选择，这样无疑会增加查找的复杂度，在这种情况下可以采用分块查找。

分块查找又称为索引顺序查找，是结合二分查找和顺序查找的一种改进方法，在分块查找里有索引表和分块的概念。索引表就是帮助分块查找的一个分块依据，其实就是一个数组，用来存储每块的最大存储值，也就是范围上限，分块就是通过索引表把数据分为几块。

在每需要增加一个元素的时候，就需要首先根据索引表，知道这个数据应该在哪一块，然后直接把这个数据加到相应的块里面，而块内的元素之间本身不需要有序。因为块内无须有序，所以分块查找特别适合元素经常动态变化的情况。

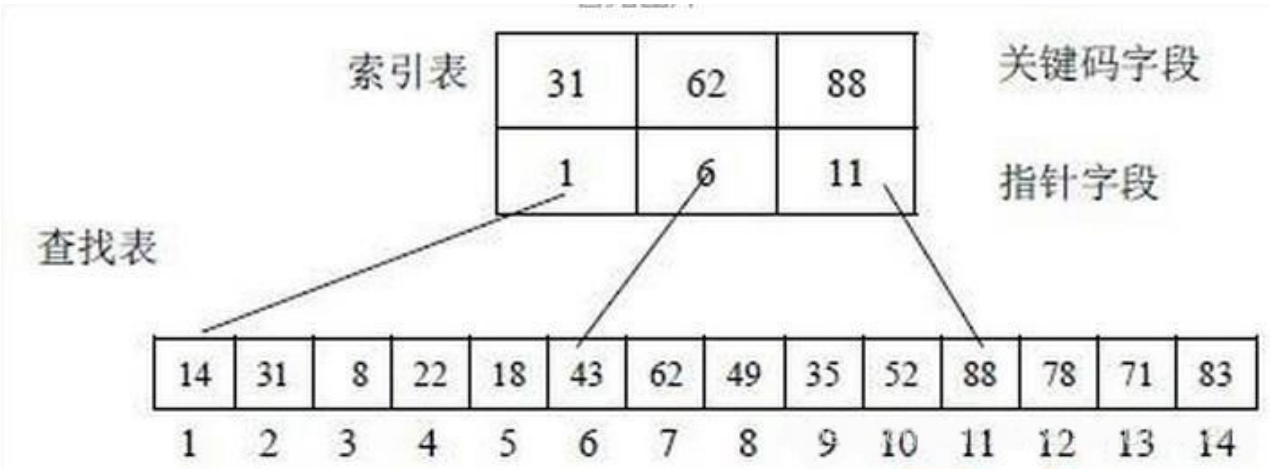
分块查找只需要索引表有序，当索引表比较大的时候，可以对索引表进行二分查找，锁定块的位置，然后对块内的元素使用顺序查找。这样的总体性能虽然不会比二分查找好，却比顺序查找好很多，最重要的是不需要数列完全有序。

分块查找要求把一个数据分为若干块，每一块里面的元素可以是无序的，但是块与块之间的元素需要是有序的。

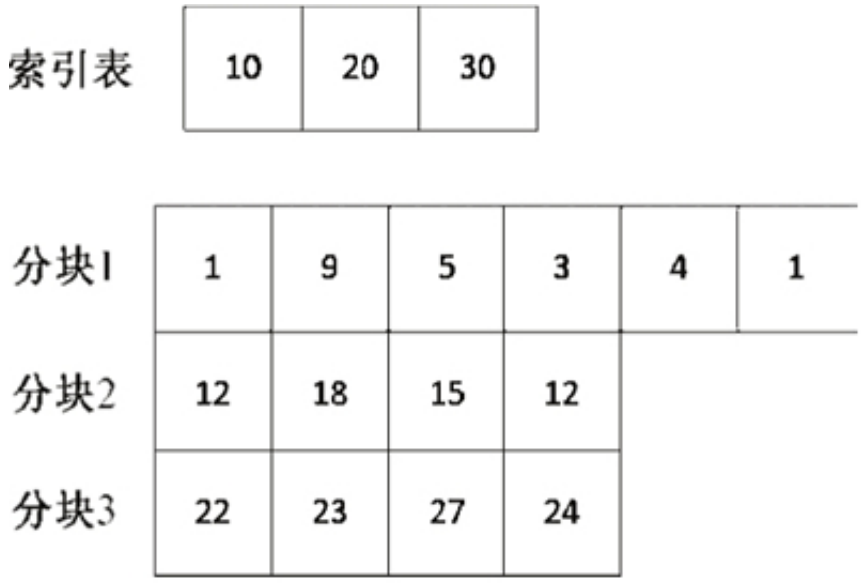
算法思想：将 n 个数据元素 “ 按块有序 ” 划分为 m 块（m<=n）。每一块中的节点不必有序，但块与块之间必须 “ 按块有序 ”；即第 1 块中任一元素的关键字都必须小于第 2 块中任一元素的关键字；而第 2 块中的任一元素又都必须小于第 3 块中的任一元素，……，以此类推。同时，分块查找需要一个索引表，用来限定每一块的范围。在增加、删除、查找元素时都需要用到。

算法流程：

1. 先选取各块中的最大关键字构成一个索引表；
2. 查找分两个部分：先对索引表进行二分查找或顺序查找，以确定待查记录在哪一块中；然后，在已确定的块中用顺序法进行查找。



所示是一个已经分好块的数据，同时有个索引表，现在要在数据中插入一个元素：



首先，看到索引表是 10、20、30，对于元素 15 来说，应该将其放在分块 2 中。于是，分块 2 的数据变为 12、18、15、12、15，直接把 15 插入分块 2 的最后就好了。

接下来就是查找操作。如果要查找上图中的 27 这个数，则首先通过二分查找索引表，发现 27 在分块 3 里，然后在分块 3 中顺序查找，得到 27 存在于数列中。

分块查找的复杂度

分块查找由于只需要索引表有序，所以特别适合用于在动态变化的数据元素序列中查找。但是如何分块比较复杂。如果分块过于稀疏，则可能导致每一块的内容过多，在顺序查找时效率很低；如果分块过密，则又会导致块数很多，无论是插入还是删除数据，都会频繁地进行二分查找；如果块数特别多，则基本上和直接二分查找的动态插入数据类似，这样分块查找就没有意义了。

所以对于分块查找来说，可以根据数据量的大小及数据的区间来进行对分块的选择。

分块查找的平均查找长度为索引查找和块内查找的平均长度之和，设索引查找和块内查找的平均查找长度分别为 L1,Ls，则分块查找的平均查找长度为：

ASL = L1+Ls

设将长度为 n 的查找表均匀的分为 b 块，每块有 s 个记录，在等概率的情况下，若在块内和索引表中均采用顺序查找，则平均查找长度为：

此时，若 s=√n，则平均查找长度取最小值：√n+1，若对索引表采用折半查找时，则平均查找长度为：

ASL = L1+Ls = log2(b+1)+(s+1)/2

时间复杂度：假设有 b 块，查询哪个块：log2b，在块中查找 n/b，总共 O(log2b+n/b)。

哈希查找

哈希查找的基本思想

算法思想：哈希的思路很简单，如果所有的键都是整数，那么就可以使用一个简单的无序数组来实现：将键作为索引，值即为其对应的值，这样就可以快速访问任意键的值。这是对于简单的键的情况，将其扩展到可以处理更加复杂的类型的键。

算法流程：

1. 用给定的哈希函数构造哈希表；
2. 根据选择的冲突处理方法解决地址冲突；
3. 在哈希表的基础上执行哈希查找。

哈希表是一个在时间和空间上做出权衡的经典例子。如果没有内存限制，那么可以直接将键作为数组的索引。那么所有的查找时间复杂度为 O(1)，如果没有时间限制，那么可以使用无序数组进行顺序查找，这样只需要很少的内存。哈希表使用了适度的时间和空间来在这两个计算之间找到了平衡。只需要调整哈希函数算法即可在时间和空间上做出取舍。

哈希查找的复杂度

单纯论查找复杂度：对于无冲突的 hash 表而言，查找复杂度为 O(1)（注意，在查找之前需要构建相应的 Hash 表）。

Hash 是一个典型以空间换时间的算法，比如原来一个长度为 100 的数组，对其查找，只需要遍历且匹配相应记录即可。从空间复杂度上来说，假如数组存储的是 byte 类型数据，那么该数组占 100 byte 空间。现在采用 Hash 算法，Hash 必须有一个规则，约束键与存储位置的关系，那么就需要一个固定长度的 hash 表，此时，仍然是 100byte 的数组，假设需要的 100byte 用来记录键与位置的关系，那么总的空间为 200bytes，而且用于记录规则的表大小会根据规则，大小可能是不定的。

* 排序算法

直接插入排序

基本思想

通常人们整理桥牌的方法是一张一张的来，将每一张牌插入到其他已经有序的牌中的适当位置。在计算机的实现中，为了要给插入的元素腾出空间，需要将其余所有元素在插入之前都向右移动一位。

直接插入的思想是：将一个记录插入到已排好序的有序表中，从而得到一个新的、记录数增 1 的有序表。

例如，排序序列（3，2，1，5）的过程是，初始时有序序列为（3），然后从位置 1 开始，先访问到 2，将 2 插入到 3 前面，得到有序序列（2，3），之后访问 1，找到合适的插入位置后得到有序序列（1，2，3），最后访问 5，得到最终有序序列（1，2，3，5）。

算法描述

一般来说，插入排序都采用 in-place 在数组上实现。具体算法描述如下：

1. 从第一个元素开始，该元素可以认为已经被排序。
2. 取出下一个元素，在已经排序的元素序列中从后向前扫描。
3. 如果该元素（已排序）大于新元素，将该元素移到下一位置。
4. 重复步骤 3，直到找到已排序的元素小于或者等于新元素的位置。
5. 将新元素插入到该位置后。
6. 重复步骤 2~5。

注意：如果比较操作的代价比交换操作大的话，可以采用二分查找法来减少比较操作的数目。该算法可以认为是插入排序的一个变种，称为二分查找插入排序。

```
/**
 * 直接插入排序
```

```
 */
public class InsertSort {

    /**
     * 通过交换进行插入排序，借鉴冒泡排序
     *
     * @param a
     */
    public static void sort(int[] a) {
        for (int i = 0; i < a.length - 1; i++) {
            // 插入前面已经排好序的序列
            for (int j = i + 1; j > 0; j--) {
                if (a[j] < a[j - 1]) {
                    int temp = a[j];
                    a[j] = a[j - 1];
                    a[j - 1] = temp;
                }
            }
        }
    }

    /**
     * 通过将较大的元素都向右移动而不总是交换两个元素
     *
     * @param a
     */
    public static void sort2(int[] a) {
        for (int i = 1; i < a.length; i++) {
            int num = a[i];
            int j;
            // 插入前面已经排好序的序列
            for (j = i; j > 0 && num < a[j - 1]; j--) {
                a[j] = a[j - 1];
            }
            a[j] = num;
        }
    }
}
```

复杂度分析

直接插入排序复杂度如下：

平均时间复杂度	最好情况	最坏情况	空间复杂度
O(n^2)	O(n)	O(n^2)	O(1)

最好情况下，当待排序序列中记录已经有序时，则需要 n-1 次比较，不需要移动，时间复杂度为 O(n)。最差情况下，待排序序列中所有记录正好逆序时，则比较次数和移动次数都达到最大值，时间复杂度为 O(n^2)，平均情况下，时间复杂度为 O(n^2)。

比较与总结

插入排序所需的时间取决于输入元素的初始顺序。例如，对一个很大且其中的元素已经有序（或接近有序）的数组进行排序将会比随机顺序的数组或是逆序数组进行排序要快得多。

希尔排序

希尔排序，也称递减增量排序算法、“ 缩小增量 ” 排序，是插入排序的一种更高效的改进版本。希尔排序是非稳定排序算法。

希尔排序是基于插入排序的以下两点性质而提出改进方法的：

- 直接插入排序在对几乎已经排好序的数据操作时，效率高，即可以达到线性排序的效率。
- 直接插入排序一般来说是低效的，因为插入排序每次只能将数据移动一位。

希尔排序是先将整个待排序的记录序列分割成为若干子序列分别进行直接插入排序，待整个序列中的记录 “ 基本有序 ” 时，再对全体记录进行依次直接插入排序。

基本思想

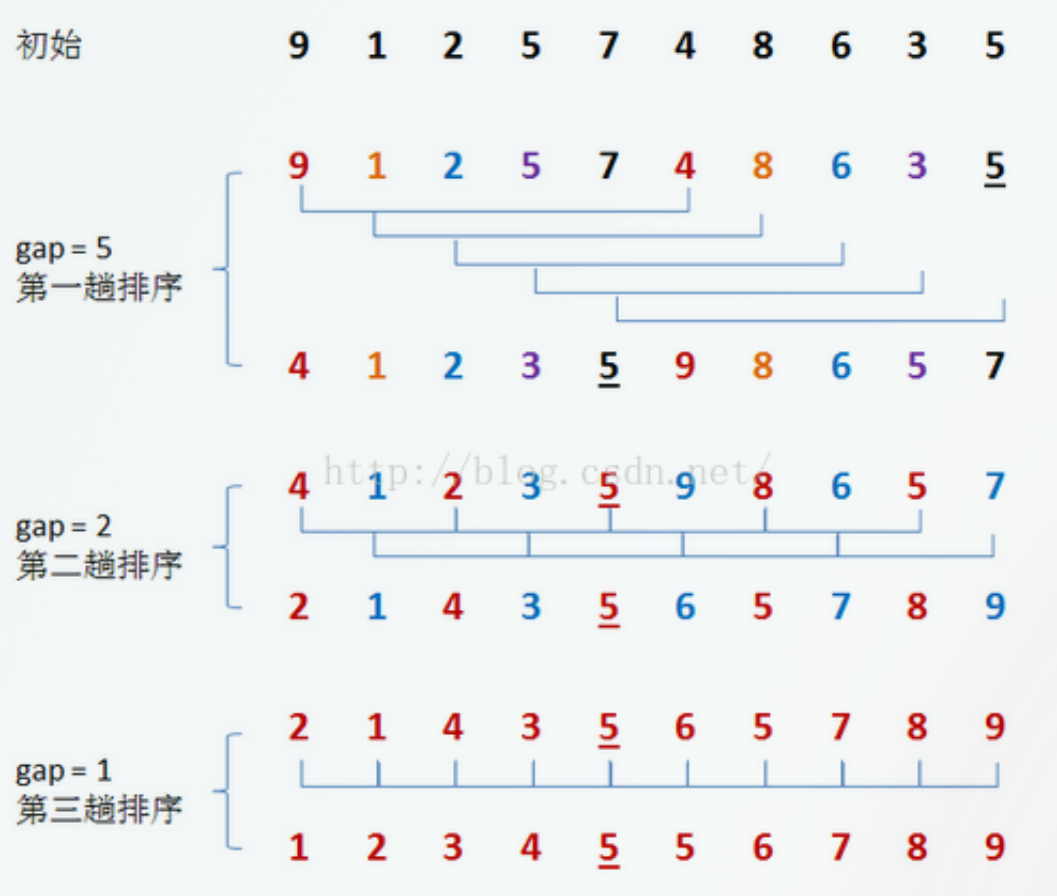
将待排序数组按照步长 gap 进行分组，然后将每组的元素利用直接插入排序的方法进行排序；每次再将 gap 折半减少，循环上述操作；当 gap = 1 时，利用直接插入，完成排序。

可以看到步长的选择是希尔排序的重要部分。只要最终步长为 1 任何步长序列都可以工作。一般来说最简单的步长取值是初次取数组长度的一半为增量，之后每次再减半，直到增量为 1。

算法描述

1. 选择一个增量序列 t1,t2,……,tk，其中 ti > tj，tk=1；
2. 按增量序列个数 k，对序列进行 k 趟排序；
3. 每趟排序，根据对应的增量 ti，将待排序序列分割成若干长度为 m 的子序列，分别对各子表进行直接插入排序。仅增量因子为 1 时，整个序列作为一个表来处理，表长度即为整个序列的长度。

效果如下：



代码实现

```
/**
 * 希尔排序
 */
public class HiliSort {
    public static void sort(int[] a) {
        int length = a.length;
        int h = 1;
        while (h < length / 3) h = 3 * h + 1;
        for (; h >= 1; h /= 3) { // 步长的循环
            System.out.println();
            System.out.println("h:" + h);
            // 分块
            for (int i = 0; i < a.length - h; i += h) {
                System.out.println("i:" + i);
                // 对分块的内容进行排序
                for (int j = i + h; j > 0; j -= h) {
                    System.out.print("j:" + j+",");
                    if (a[j] < a[j-h]) {
                        int temp = a[j];
                        a[j] = a[j - h];
                        a[j - h] = temp;
                    }
                }
                System.out.println();
```

```
            }
        }
    }

    public static void main(String[] args) {
        int a[] = new int[]{9, 1, 2,  7, 4, 8, 6, 3, 5};
        sort(a);
        System.out.println("a:" + Arrays.toString(a));
    }
}
```

简单选择排序

堆排序

冒泡排序

归并排序

基数排序

快速排序

在平均状况下，排序 n 个项目要 O(nlogn) 次比较。在最坏状况下则需要 O(n^2) 次比较，但这种状况并不常见。事实上，快速排序通常明显比其他 O(nlogn) 算法更快，因为它的内部循环（inner loop）可以在大部分的架构上很有效率地被实现出来。

基本思想

快速排序的基本思想：挖坑填数 + 分治法。

快速排序又是一种分治法思想在排序算法上的典型应用。本质上来看，快速排序应该算是在冒泡排序基础上的递归分治法，是对冒泡排序的一种改进。

快速排序的主要思想是：在待排序的序列中选择一个称为主元的元素，将数组分为两部分，使得第一部分中的所有元素都小于或等于主元，而第二部分中的所有元素都大于主元，然后对两部分递归地应用快速排序算法，整个排序过程可以递归进行，以此达到整个数据变成有序序列。

算法描述

快速排序使用分治策略来把一个序列（list）分为两个子序列（sub-list）。步骤为：

1. 从数列中挑出一个元素，称为 “ 基准 ”（pivot）。
2. 重新排序数列，所有比基准值小的元素排放在基准前面，所有比基准值大的元素摆在基准后面（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。
3. 递归地（recursively）把小于基准值元素地子数列和大于基准值元素的子数列排序。

递归到最底部时，数列的大小是零或一，也就是已经排序好了。这个算法一定会结束，因为在每次的迭代（iterator）中，它至少会把一个元素摆到它最后的位置去。

代码实现

用伪代码描述如下：

1. i = L;j = R; 将基准数挖出形成第一个坑 a[i]。
2. j--，由后向前找比它小的数，找到后挖出此数填前一个坑 a[i]中。
3. i++，由前向后找比它大的数，找到后也挖出此数填到前一个坑 a[j] 中。
4. 再重复执行 2、3 二步，直到 i==j，将基准数填入 a[i] 中。

```
    public static void sort(int[] a, int low, int high) {
        // 已经排完
        if (low >= high) {
            return;
        }
        int left = low;
        int right = high;
        // 保存基准位置
        int pivot = left;
        while (left < right) {
            // 从后往前找到比基准小的元素
            while (left < right && a[right] >= a[pivot]) {
                right--;
            }
            // 从前往后找到比基准大的元素
            while (left < right && a[left] <= a[pivot]) {
                left++;
            }
            if (left < right) {
                int temp = a[left];
                a[left] = a[right];
                a[right] = temp;
            }
        }
        // 放值基准值
        int temp = a[left];
        a[left] = a[pivot];
        a[pivot] = temp;
```

```
        // 分支递归快排
        sort(a, low, left - 1);
        sort(a, left + 1, high);
    }
```

上面是递归版的快速排序：通过把基准插入到合适的位置来实现分治，并递归地对分治后的两个划分继续快排。那么非递归版的快排如何实现呢?

因为递归的本质是栈，所以非递归实现的过程中，可以借助栈来保存中间变量就可以实现非递归了。在这里中间变量也就是通过 Pritation 函数划分取键之后分成左右两部分的首尾指针，只需要保存这两部分的首尾指针即可。

复杂度分析

在快速排序算法中，比较关键的一个部分是主元的选择。在最差情况下，划分由 n 个元素构成的数组需要进行 n 次比较和 n 次移动，因此划分需要的时间是 O(n)。在最差情况下，每次主元会将数组划分为一个大的子数组和一个空数组，这个大的子数组的规模是在上次划分的子数组的规模上减 1，这样在最差情况下算法需要 (n-1)+(n-2)+...+1=O(n^2) 时间。

最佳情况下，每次主元将数组划分为规模大致相同的两部分，时间复杂度为 O(nlogn)。

以下是快速排序算法复杂度：

平均时间复杂度	最好情况	最坏情况	空间复杂度
O(nlog2 n)	O(n log2 n)	O(n^2)	O(1)(原地分区递归版)