目录

1. 排序算法有哪些？

冒泡排序、快速排序、选择排序、插入排序、堆排序算法

2. 最快的排序算法是哪个？

7. 计数排序

我们希望能线性的时间复杂度排序，如果一个一个比较，显然是不实际的，书上也在决策树模型中论证了，比较排序的情况为 `nlogn` 的复杂度。既然不能一个一个比较，我们想到一个办法，就是如果**在排序的时候就知道他的位置，那不就是扫描一遍，把他放入他应该的位置**不就可以了。 要知道**他的位置，我们只需要知道有多少不大于他不就可以了**吗？

7.1 性能分析

最好，最坏，平均的时间复杂度 `O(n+k)`，天了噜， 线性时间完成排序，且稳定。

优点：不需要比较函数，利用地址偏移，对范围固定在[0,k]的整数排序的最佳选择。是排序字节串**最快的排序算法**。

缺点：由于用来计数的数组的长度取决于待排序数组中数据的范围（等于待排序数组的最大值与最小值的差加上1），这使得计数排序对于数据范围很大的数组，需要大量时间和内存。

7.2 核心代码

```
public int[] countsort(int A[]){
    int[] B = new int[A.length]; //to store result after sorting
    int k = max(A);
    int [] C = new int[k+1]; // to store temp
    for(int i=0;i<A.length;i++){
        C[A[i]] = C[A[i]] + 1;
    }
    // 小于等于A[i]的数的有多少个，存入数组C
    for(int i=1;i<C.length;i++){
        C[i] = C[i] + C[i-1];
    }
    //逆序输出确保稳定-相同元素相对顺序不变
    for(int i=A.length-1;i>=0;i--){

        B[C[A[i]]-1] = A[i];
        C[A[i]] = C[A[i]]-1;
    }
    return B;
}
```

3. 手写一个冒泡排序

冒泡排序的**算法实现**如下:【排序后，数组从小到大排列】

```
 /*
 * 冒泡排序
 * 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
 * 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。
 * @param numbers 需要排序的整型数组
 */
public static void bubbleSort(int[] numbers)
{
    int temp = 0;
    int size = numbers.length;
    for(int i = 0 ; i < size-1; i ++)
    {
    for(int j = 0 ;j < size-1-i ; j++)
    {
        if(numbers[j] > numbers[j+1])  //交换两数位置
        {
        temp = numbers[j];
        numbers[j] = numbers[j+1];
        numbers[j+1] = temp;
        }
    }
    }
}
```

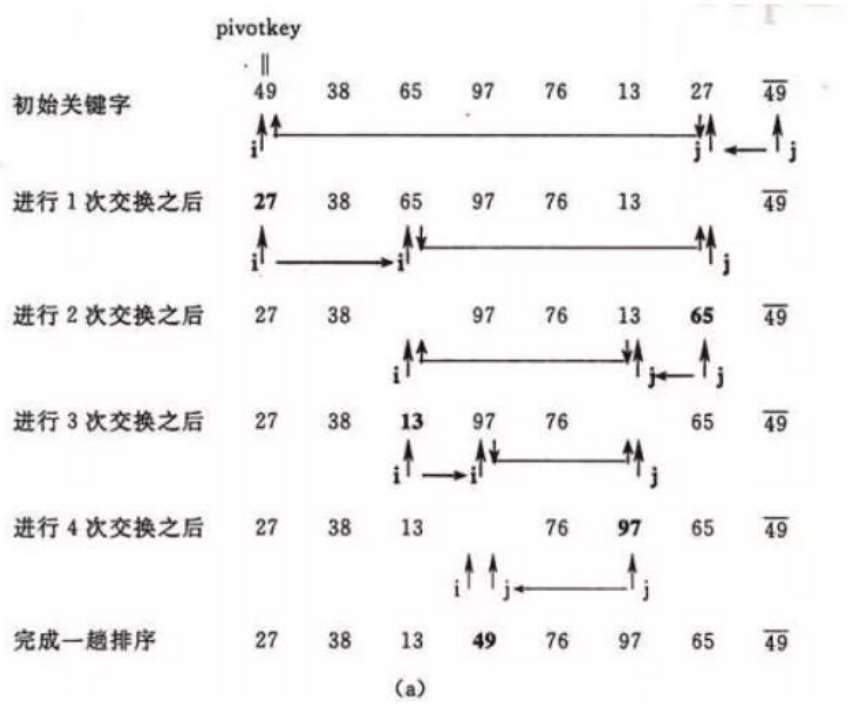
4. 手写快速排序代码

快速排序的基本思想：

通过一趟排序将待排序记录分割成独立的两部分，其中一部分记录的关键字均比另一部分关键字小，则分别对这两部分继续进行排序，直到整个序列有序。

快速排序的**示例**：

（a）一趟排序的过程：



（b）排序的全过程：

1.查找中轴（最低位作为中轴）所在位置：

```
 /**
 * 查找出中轴（默认是最低位low）的在numbers数组排序后所在位置
 *
 * @param numbers 带查找数组
 * @param low   开始位置
 * @param high  结束位置
 * @return  中轴所在位置
 */
public static int getMiddle(int[] numbers, int low,int high)
{
    int temp = numbers[low]; //数组的第一个作为中轴
    while(low < high)
    {
    while(low < high && numbers[high] >= temp)
    {
        high--;
    }
    numbers[low] = numbers[high];//比中轴小的记录移到低端
    while(low < high && numbers[low] < temp)
    {
        low++;
    }
    numbers[high] = numbers[low] ; //比中轴大的记录移到高端
    }
    numbers[low] = temp ; //中轴记录到尾
    return low ; // 返回中轴的位置
}
```

2、 递归形式的分治排序算法：

```
 /**
 *
 * @param numbers 带排序数组
 * @param low  开始位置
 * @param high 结束位置
 */
public static void quickSort(int[] numbers,int low,int high)
{
    if(low < high)
    {
        int middle = getMiddle(numbers,low,high); //将numbers数组进行一分为二
        quickSort(numbers, low, middle-1);   //对低字段表进行递归排序
        quickSort(numbers, middle+1, high); //对高字段表进行递归排序
    }

}
```

3、快速排序提供方法调用：

```
 /**
 * 快速排序
 * @param numbers 带排序数组
 */
public static void quick(int[] numbers)
{
    if(numbers.length > 0)   //查看数组是否为空
    {
    quickSort(numbers, 0, numbers.length-1);
    }
}
```

5. 快速排序的过程、时间复杂度、空间复杂度

6. 堆排序过程、时间复杂度及空间复杂度

7. 写出你所知道的排序算法及时空复杂度，稳定性

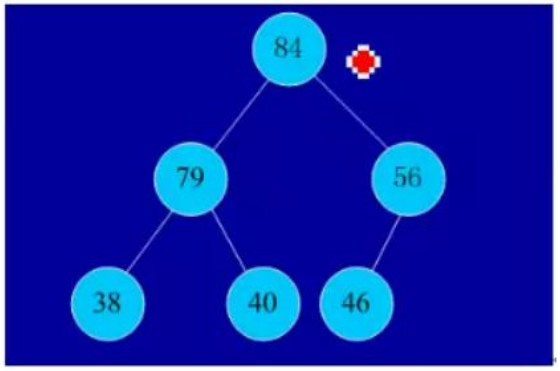
各种常用排序算法						
类别	排序方法	时间复杂度			空间复杂度	稳定性
		平均情况	最好情况	最坏情况	辅助存储	
插入排序	直接插入	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	shell排序	$O(n^{1.3})$	$O(n)$	$O(n^2)$	$O(1)$	不稳定
选择排序	直接选择	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
	堆排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(1)$	不稳定
交换排序	冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	快速排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(n^2)$	$O(nlog_2n)$	不稳定
归并排序		$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(1)$	稳定
基数排序		$O(d(r+n))$	$O(d(n+rd))$	$O(d(r+n))$	$O(rd+n)$	稳定
注：基数排序的复杂度中，r代表关键字的基数，d代表长度，n代表关键字的个数						

8. 手写堆排序

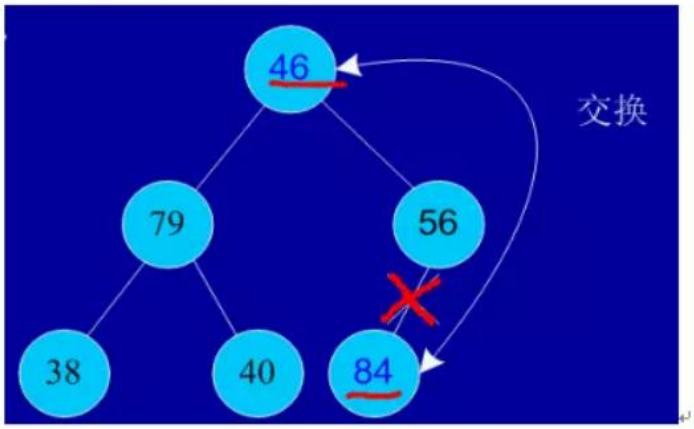
2、实例：

初始序列：46,79,56,38,40,84

建堆：



交换，从堆中踢出最大数：



剩余结点再建堆，再交换踢出最大数

依次类推：最后堆中剩余的最后两个结点交换，踢出一个，排序完成。

```java
public class HeapSort {
    public static void main(String[] args) {
        int[] a={49,38,65,97,76,13,27,49,78,34,12,64};
        int arrayLength=a.length;
        //循环建堆
        for(int i=0;i<arrayLength-1;i++){
            //建堆
            buildMaxHeap(a,arrayLength-1-i);
            //交换堆顶和最后一个元素
            swap(a,0,arrayLength-1-i);
            System.out.println(Arrays.toString(a));
        }
    }
    //对data数组从0到lastIndex建大顶堆
    public static void buildMaxHeap(int[] data, int lastIndex){
         //从lastIndex处节点（最后一个节点）的父节点开始
        for(int i=(lastIndex-1)/2;i>=0;i--){
            //k保存正在判断的节点
            int k=i;
            //如果当前k节点的子节点存在
            while(k*2+1<=lastIndex){
                //k节点的左子节点的索引
                int biggerInd
                //如果biggerIndex小于lastIndex，即biggerIndex+1代表的k节点的右子节点存在
                if(biggerIndex<lastIndex){
                    //若果右子节点的值较大
                    if(data[biggerIndex]<data[biggerIndex+1]){
                        //biggerIndex总是记录较大子节点的索引
                        biggerIndex++;
                    }
                }
                //如果k节点的值小于其较大的子节点的值
                if(data[k]<data[biggerIndex]){
                    //交换他们
                    swap(data,k,biggerIndex);
                    //将biggerIndex赋予k，开始while循环的下一次循环，重新保证k节点的值大
                    k=biggerIndex;
                }else{
                    break;
                }
            }
        }
    }
    //交换
    private static void swap(int[] data, int i, int j) {
        int tmp=data[i];
        data[i]=data[j];
        data[j]=tmp;
    }
}
```

9. 二叉树给出根节点和目标节点，找出从根节点到目标节点的路径

1。 首先是检测某个节点时候在某个二叉树中出现过。

```
/*
// If the tree with head pHead has a node pNode, return true.
// Otherwise return false.
*/
bool HasNode(TreeNode* pHead, TreeNode* pNode)
{
        if(pHead == pNode)
                return true;
        bool has = false;
        if(pHead->m_pLeft != NULL)
                has = HasNode(pHead->m_pLeft, pNode);
        if(!has && pHead->m_pRight != NULL)
                has = HasNode(pHead->m_pRight, pNode);
        return has;
}
```

2. 从根节点到某一个节点的路径

```
/*
// Get the path form pHead and pNode in a tree with head pHead
*/
bool GetNodePath(TreeNode* pHead, TreeNode* pNode, std::list<TreeNode*>& path)
{
        if(pHead == pNode)
                return true;

        path.push_back(pHead);

        bool found = false;
        if(pHead->m_pLeft != NULL)
                found = GetNodePath(pHead->m_pLeft, pNode, path);
        if(!found && pHead->m_pRight)
                found = GetNodePath(pHead->m_pRight, pNode, path);
        if(!found)
                path.pop_back();
        return found;
}
```

10. 给阿里 2 万多名员工按年龄排序应该选择哪个算法？

11. GC 算法(各种算法的优缺点以及应用场景)

https://www.jianshu.com/p/8c915179fd02

12. 蚁群算法与蒙特卡洛算法

13. 子串包含问题(KMP 算法)写代码实现

14. 一个无序，不重复数组，输出 N 个元素，使得 N 个元素的和相加为 M，给出时间复杂度、空间复杂度。手写算法

15. 万亿级别的两个 URL 文件 A 和 B，如何求出 A 和 B 的差集 C(提示：Bit 映射->hash 分组->多文件读写效率->磁盘寻址以及应用层面对寻址的优化)

16. 百度 POI 中如何试下查找最近的商家功能(提示:坐标镜像+R 树)。

17. 两个不重复的数组集合中，求共同的元素。

18. 两个不重复的数组集合中，这两个集合都是海量数据，内存中放不下，怎么求共同的元素？

19. 一个文件中有 100 万个整数，由空格分开，在程序中判断用户输入的整数是否在此文件中。说出最优的方法

20. 一张 Bitmap 所占内存以及内存占用的计算

21. 2000 万个整数，找出第五十大的数字？

22. 烧一根不均匀的绳，从头烧到尾总共需要 1 个小时。现在有若干条材质相同的绳子，问如何用烧绳的方法来计时一个小时十五分钟呢？

23. 求 1000 以内的水仙花数以及 40 亿以内的水仙花数

24. 5 枚硬币，2 正 3 反如何划分为两堆然后通过翻转让两堆中正面向上的硬 8 币和反面向上的硬币个数相同

25. 时针走一圈，时针分针重合几次

26. N*N 的方格纸,里面有多少个正方形

27. x 个苹果，一天只能吃一个、两个、或者三个，问多少天可以吃完？