**Java算法总结**

**字符串**

**1.字符串翻转**

给定一个字符串，长度为n，要求把字符串前面的m个字符移动到字符串的尾部，如把字符串“abcdef”前面的2个字符'a'和'b'（**m=2**）移动到字符串的尾部，使得原字符串变成字符串“cdefab”。  
要求：请写一个函数完成此功能，要求对长度为n的字符串操作的时间复杂度为 O(n)，空间复杂度为 O(1)。

**常规思路：**  
1）取前面第i（i >= 1 && i <= m）个字符，然后将后面的m-1个字符移动到前面，字符串第n个位置放第i个字符；  
2）i加1，然后重复步骤1，直到i==m时旋转完成。

则上面的例子经过下面两步骤可以完成：  
**步骤一：ab|cdef -> bcdef|a**  
**步骤二：b|cdefa -> cdef|ab**  
针对长度为n的字符串来说，假设需要移动m个字符到字符串的尾部，那么总共需要 m x n 次操作，同时设立一个变量保存第一个字符，这样，时间复杂度为O(m \* n)，空间复杂度为O(1)。

**反转法**：  
思路：字符串分成XY两部分，我们可以先对X、Y反转，然后对X^Y^进行一次反转的到YX。这样，我们可以达到题目时间复杂度和空间复杂度的限定条件。  
实例：  
**步骤一：ab|cdef -> ba|fedc**  
**步骤二：ba|fedc -> cdef|ab**

算法实现:

/\*\*

\* 字符串翻转

\* 先局部翻转,然后整体翻转

\*/

public static void rotateStr(char[] strArray, int m){

if(strArray == null) return;

if(m < 0 || m >= strArray.length) return;

revertPartition(strArray, 0, m);

revertPartition(strArray, m+1, strArray.length -1);

revertPartition(strArray, 0, strArray.length -1);

}

public static void revertPartition(char[] strArray, int start, int end){

while(start < end){

char tmp = strArray[start];

strArray[start++] = strArray[end];

strArray[end--] = tmp;

}

}

测试代码：

public static void main(String[] args){

char[] strArray = {'a', 'b', 'c', 'd','e','f'};

rotateStr(strArray, 1);

System.out.println(String.valueOf(strArray));

System.exit(0);

}

**说明：测试代码中，m是取值范围是 0 到 n-1，n是字符数组的长度。**

**题目延伸：**句子中单词的翻转，如输入i am a student，输出student a am i。  
思路：先对句子中的每个单词进行翻转，然后对整个句子做一次翻转。  
算法实现：

/\*\*

\* 句子翻转

\* 先局部翻转单词,然后整体翻转句子

\*/

public static void rotateSentence(char[] sentence){

if(sentence == null) return;

int start = 0, end = 0;

for(;end<= sentence.length && start < sentence.length; end++){

//如果end遍历到空格或者达到句子末尾，则找到一个单词并进行翻转

if(( end < sentence.length && sentence[end] == 32) || end == sentence.length){

revertPartition(sentence, start, end-1);

start = end + 1;

}

}

//整个句子进行翻转

revertPartition(sentence, 0, sentence.length -1);

}

测试代码：

public static void main(String[] args){

char[] sentence = {'i', ' ', 'a', 'm', ' ', 'a',' ','s','t','u','d','e','n','t'};

rotateSentence(sentence);

System.out.println(String.valueOf(sentence));

System.exit(0);

}

**2.字符串包含**

给定两个分别由字母组成的字符串A和字符串B，字符串B的长度比字符串A短。请问，如何最快地判断字符串B中所有字母是否都在字符串A里？  
下面，我们给出两种较忧的解法：  
**计数排序：**  
假设有一个仅由字母组成字串，让每个字母与一个素数对应，从2开始，往后类推，A对应2，B对应3，C对应5......。遍历第一个字串，把每个字母对应素数相乘，最终，会得到一个整数。  
利用上面字母和素数的对应关系，对第二个字符串进行遍历，用上面的整数除以该字母对应的素数。如果有余数，则字符串B的字母不都在字符串A中；如果整个过程中没有余数，则字符串B的字母都在字符串A中。

算法实现：

//此方法只有理论意义，因为整数乘积很大，有溢出风险

public static boolean contain(char[] a, char[] b) {

int[] p = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101};

int multi = 1;

for (int i = 0; i < a.length; ++i) {

int item = p[a[i] - 'a'];

if (multi % item != 0) {

multi \*= item;

}

}

for (int i = 0; i < b.length; ++i) {

int item = p[b[i] - 'a'];

if (multi % item != 0) {

return false;

}

}

return true;

}

**位运算**:  
我们可以用26个bit位分别来表示26个字母，对字符串A，用位运算计算出一个“签名”，遍历字符串B中的字母，看在A中是否存在。

算法实现：

public static boolean bitContain(char[] a, char[] b) {

int hash = 0;

for (int i = 0; i < a.length; i++) {

hash |= 1 << (a[i] - 'a');

}

for (int i = 0; i < b.length; i++) {

if ((hash & 1 << (b[i] - 'a')) == 0) {

return false;

}

}

return true;

}

**位运算的扩展：给定一个int类型的整数 ，输出它的二进制数中 1的个数**。

算法实现：

int input = 9821;

int num = 0;

while(input != 0){

input = input & (input-1);

num ++;

}

**2.字符串全排列**

输入一个字符串（**不存在相同的字符**），打印出该字符串中字符的所有排列。  
**递归实现：**  
从字符串中依次选出每一个元素，作为排列的第一个元素，然后对剩余的字符串进行全排列，如此递归处理，从而得到字符串的全排列。  
例子：输出abc的全排列  
1）a固定在第一个位置，求b、c的排列，得到bc，cb，整个字符串的全排列有abc，acb。

2）对于原始字符串abc，a和b交换位置得到字符串bac，b固定在第一个位置，求a、c的排列，得到ac，ca，整个字符串的全排列有bac，bac。

3）对于原始字符串abc，a和c交换位置得到字符串cab，c固定在第一个位置，求a、b的排列，得到ab，ba，整个字符串的全排列有cab，cba。

算法实现：

//字符串的全排列

public static void perSort(char[] arrayStr, int from, int to) {

if (to <= 1) {

System.out.println(String.valueOf(arrayStr));

return;

}

if(from == to){

System.out.println(String.valueOf(arrayStr));

}

else {

for (int i = from; i <= to; i++) {

swap(arrayStr, i, from);

perSort(arrayStr, from + 1, to);

swap(arrayStr, i, from);

}

}

}

上述递归算法的**时间复杂度是O(n!)**。

**数组**

**1. 给定一个长度为n的数组，输出最小的k个数。**

下面我们给出三种解题思路：  
1）输出最小的k个数，我们很容易想到对数组序列进行一次排序，然后输出前k个元素，假设排序用的快排，则时间复杂度是**O(nlogn + k)**。

2）题目没有要求最小的k个数有序，也没要求最后n-k个数有序。既然这样，就没有必要对所有元素进行排序。我们知道选择排序是选择最小的元素放到第一个位置，次小的元素放到第二个位置......n小的元素放到第n个位置。  
这里，我们要找的是最小的k个数，所以只需要进行k次选择，然后输出前k个数就可以，时间复杂度是O(nk)。

3）快排的原理是先选择一个基准元素，经过一次partition后，基准元素前面的元素都是比它大的，后面的元素都是比它小的。我们可以根据快排的这一特性来解这道题，假设经过一次partition后，基准元素在数组中的位置是middle，接下来可以分以下三种情况考量：

**情况一：k - 1 = middle，说明middle正是我们要找的第k小的元素，且从0到k-1是序列最小的k个数；**  
**情况二：k - 1 < middle，说明我们要找的第k小的元素在数列的0到middle-1中，我们需要递归地进行一次划分查找；**  
**情况三：k - 1 > middle，说明我们要找的第k小的元素在数列的middle+1到n-1中，我们需要递归地进行一次划分查找。**

**算法实现**

/\*\*

\* 给定一个整形数组序列,找到最小的n个数

\*/

public static void findMinK(int[] element, int start, int end, int k){

if(end <= k -1) return;

if(start < end){

int middle = partition(element, start, end);

if(k-1 == middle) print(element, k);

else if(k-1 < middle) findMinK(element, start, middle -1, k);

else findMinK(element, middle +1 , end, k);

}

}

public static int partition(int[] element, int low, int high) {

int baseElement = element[low];

while (low < high) {

while (low < high && baseElement <= element[high]) high--;

element[low] = element[high];

while (low < high && baseElement >= element[low]) low++;

element[high] = element[low];

}

element[low] = baseElement;

return low;

}

**2.数组序列中有一个数字出现的次数超过了数组长度的一半，请找出这个数字。**

思路一：  
如果数组序列无序，那么我们可以先把数组中所有这些数字先进行排序（至于排序方法可选取最常用的快速排序）。一个数字在数组中的出现次数超过了一半，那么在已排好序的数组索引的N/2处（从零开始编号），就一定是这个数字。因此，排完序后直接输出数组中的第N/2处的数字即可，这个数字即是整个数组中出现次数超过一半的数字，总的时间复杂度是 **O(n\*logn)**。

思路二：  
我们能不能换种思路考虑上面这道题，对于这个数组序列，我们每次去掉两个不同的数，直到最后剩下的数就是我们想找的数。其实，我们完全不用每次去掉两个数，我们只需声明两个变量num，time，其中num分别表示当前遍历的某个数和这个数出现的次数：

1）num初始化为第一个数，time初始化为1；

2）遍历下一个数，如果下一个数和num相等，则time加1，不等，time减1；

3）如果time等于0，则num初始化下一个待遍历的数，同时time初始化为1，直到遍历完成，num并是我们要找的数。

算法实现：

public static void findHalfNum(int[] element) {

if (element.length < 1) return;

int num = element[0], time = 1;

for (int i = 0; i < element.length; i++) {

if (time == 0) {

num = element[i];

time = 1;

}

if (num == element[i]) {

time++;

} else {

time--;

}

}

System.out.println("数组中超过一半的数是:" + num);

}

上述算法的时间复杂度是：**O(n)**。

**二叉树**

**1.二叉树反转**

给你一个二叉树，将二叉树的左右子树进行交换（不使用递归）。  
递归算法：

/\*\*

\* 递归实现

\* @param curNode:输入节点

\*/

public static void revertRecursion(Node curNode) {

if (curNode == null) return;

//当前节点非空,交换左右子树

swap(curNode);

revertRecursion(curNode.left);

revertRecursion(curNode.right);

}

非递归算法，用Stack实现：

public static void revertStack(Node root){

Stack<Node> nodeStack = new Stack<>();

if(root != null) nodeStack.push(root);

while(!nodeStack.isEmpty()){

Node curNode = nodeStack.pop();

swap(curNode);

if(curNode.left != null) nodeStack.push(curNode.left);

if(curNode.right != null) nodeStack.push(curNode.right);

}

}

附：Node类、swap方法。  
Node类:

public static class Node{

public Node left;

public Node right;

public Object data;

}

swap方法:

public static void swap(Node curNode){

Node tmp = curNode.left;

curNode.left = curNode.right;

curNode.right = tmp;

}

**单例模式**

单例模式最优方案： 线程安全，并且效率高，代码如下：

public class Singleton {

//使用volatile保证了多线程访问时instance变量的可见性

private volatile static Singleton instance;

// 定义一个私有构造方法

private Singleton() {

}

public static Singleton getInstance() {

// 对象实例化时与否判断（不使用同步代码块，instance不等于null时，直接返回对象，提高运行效率)

if (instance == null) {

//同步代码块（对象未初始化时，使用同步代码块，保证多线程访问时对象在第一次创建后，不再重复被创建）

synchronized (Singleton.class) {

//未初始化，则初始instance变量

if (instance == null) {

instance = new Singleton();

}

}

}

return instance;

}

}

### 冒泡排序

1.自我理解

* 1．比较相邻的前后二个数据，如果前面数据大于后面的数据，就将二个数据交换。
* 2．这样对数组的第0个数据到N-1个数据进行一次遍历后，最大的一个数据就“沉”到数组第N-1个位置。
* 3．N=N-1，如果N不为0就重复前面二步，否则排序完成。

2.代码

public static void bubbleSort(int[] source) {

int length = source.length;

for (int i = 0; i < length - 1; i++) {

for (int j = 0; j < length - 1 - i; j++) {

if (source[j] > source[j + 1]) {

swap(source, j, j + 1);

}

}

}

printArray(source);

}

public static void swap(int[] source, int x, int y) {

int temp = source[x];

source[x] = source[y];

source[y] = temp;

}

public static void printArray(int[] source) {

for (int i = 0; i < source.length; i++) {

System.out.print("\t" + source[i]);

}

}

### 选择排序

1.自我理解

* 1.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。
* 2.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。
* 3.以此类推，直到所有元素均排序完毕。

2.代码

public static void selectSort(int[] source) {

int length = source.length;

int minIndex = 0;

for (int i = 0; i < length - 1; i++) {

minIndex = i;

for (int j = i + 1; j < length; j++) {

if (source[j] < source[minIndex]) {

minIndex = j;

}

}

if (minIndex != i) {

swap(source, minIndex, i);

}

}

printArray(source);

}

### 快速排序

1.自我理解

**挖坑填数+分治法**

2.代码

public static int[] quickSort(int[] source, int l, int r) {

if (l < r) {

int i = l, j = r, x = source[l];

while (i < j) {

while (i < j && source[j] >= x)

j--;

if (i < j) {

source[i++] = source[j];

}

while (i < j && source[i] < x)

i++;

if (i < j) {

source[j--] = source[i];

}

}

source[i] = x;

quickSort(source, l, i - 1);

quickSort(source, i + 1, r);

}

return source;

}

### 总结

