### $1\_testFN

以下示例：为证明当 时间复杂度一致时，比的不是常数项。，而是**常数项所运行的时间**，因为你不清楚它里面是使用的位运算还是算数运算，而位运算要快于算数运算。

**public static void process1() {**

**int a = 1;**

**int x = 1000;**

**for (int i = 0; i < x; i++) {**

**a = 2 + 5;**

**a = 4 \* 7;**

**}**

**}**

**public static void process2() {**

**int a = 1;**

**int x = 10000;**

**for (int i = 0; i < x; i++) {**

**a = 3 | 6;**

**a = 3 & 4;**

**}**

**}**

### $2\_选择排序

时间复杂度O(n^2),额外空间复杂度O(1)。 **找到最小值，换位置**

**public static void process1(int[] arr) {**

**if (arr == null || arr.length <= 1) {**

**return;**

**}**

// 1、在i到n-1中找最小值，然后把最小值放到i位置上去，下标0到N-1，其实就是第一个到最后一个

// 2、找到最小值的下标和当前处理的下标为i的数据交换位置

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

// 3、在整个过程中，开辟的空间有i,j,minIndex，且minIndex，用完即释放，有限变量来维持此算法，所以额外空间复杂度是O(1)

**int minIndex = i;**

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

**if (arr[j] < arr[minIndex]) {**

**minIndex = j;**

**}**

**}**

**swap(arr, i, minIndex);**

**}**

**}**

**private static void swap(int[] arr, int i, int j) {**

**if (j != i) {**

**// int temp = arr[i];**

**// arr[i] = arr[minIndex];**

**// arr[minIndex] = temp;**

**arr[i] = arr[i] ^ arr[j];**

**arr[j] = arr[i] ^ arr[j];**

**arr[i] = arr[i] ^ arr[j];**

**}**

**}**

**public static void main(String[] args) {**

**int[] arr = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 99, 0, 33, 5};**

**process1(arr);**

**System.err.println(Arrays.toString(arr));**

**}**

### $3\_冒泡排序

**相邻位置比大小，大的向右移动，每迭代一次，都能找到当前最大值，即依次确定的位置为 N，N-1，N-2，依次类推**。从大到小.时间复杂度O(n^2),额外空间复杂度O(1)。

**public static void process1(int[] arr) {**

**if (arr == null || arr.length <= 1) {**

**return;**

**}**

**// 1、比较N次**

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

**// 2、第一次比较数组所有内容，两两交换，确定最后一个数字是最大的那个**

**// 3、第二次比较除去最后一个数字的，两两交换，确定倒数第二大数字**

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

**if (arr[j] > arr[j + 1]) {**

**swap(arr, j, j + 1);**

**}**

**}**

**}**

**}**

**private static void swap(int[] arr, int i, int j) {**

**if (j != i) {**

**// int temp = arr[i];**

**// arr[i] = arr[j];**

**// arr[j] = temp;**

**// ^运算，不同为1相同为0的计算方式，无进位相加**

**// a: 10110**

**// b: 00111**

**// 和：10001 满二不进1**

**// int a = 甲**

**// int b = 乙**

**// a = a ^ b; a=甲^乙 b=乙**

**// b = a ^ b; b=甲^乙^乙=甲^0=甲 a=甲^乙**

**// a = a ^ b; a=甲^乙^甲=乙^0=乙 b=甲**

**// 前提条件：i和j不能用同一块内存区域**

**arr[i] = arr[i] ^ arr[j];**

**arr[j] = arr[i] ^ arr[j];**

**arr[i] = arr[i] ^ arr[j];**

**}**

**}**

### $4\_找1种出现了奇数次的数

**要求时间复杂度 O（N）**，空间复杂度O（1） 找到数组中出现奇数次的一个数 思路： 申明 eor = 0,将数组中的每一个元素与EOR进行^运算，最后剩下的就是奇数次的那个数 原因：**出现偶数次，则意味着相互与运算 = 0（因为与运算中，不同为1，相同为0）， 例如 2,2,3,3,3,1,1 所以 2,2=0 1,1=0 3^3^3=3 所以所有偶数与运算后结果为1，所以最终结果就是奇数次的那个数**

**public static int process1(int[] arr) {**

**int eor = 0;**

**// for (int i : arr) {**

**// eor ^= i;**

**// }**

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

**eor ^= arr[i];**

**}**

**return eor;**

**}**

### $5\_找2种出现了奇数次的数

**找出数组中，出现奇数次的，两个数 假设数组**中，出现奇数次的有两个数，例如 1,1,2,3,4,3,2,5,5,5 设：eor=0，假设那两个数为a和b，继续 ^ 后可得 eor = a^b 又因为是两个数所以a!=b,所以eor=a^b就一定不等于0 所以eor的二进制表示中，一定有一位数不等于0，假设这是第8位，那么就使用 变量eor2去亦或数组中每个元素的第8位是1的那些数，出现偶数次的依旧不做干扰，此时剩下的就是a或者b 然后 eor ^ eor2 = a^b^a = b,就可以 知道这俩数是多少。

**public static void process1(int[] arr) {**

**//eor = a^b eor!=0**

**int eor = 0;**

**for (int i : arr) {**

**eor ^= i;**

**}**

//eor必然有一个位置上是1，因为a!=b,所以a^b后，二进制为1的数，必然a和b二进制位所在的该位置不相同

// a： 1000101

// b： 0010111

// a^b: 1010010 (相同为0，不同为1)

//此时就可以，提取出eor最右边的1，也就是说在此位上 a一定不等于b

// eor: 1010111100

// 取反eor: 0101000011

// 取反eor+1： 0101000100

// eor & (取反eor+1) = 0000000100 (&运算，如果两个对应的位都是1，则结果位为1；否则为0)

// ==================================================

//  **提取最右侧的1**

**int rightOne = eor & (~eor + 1);**

// ==================================================

**int onlyOne = 0;**

**for (int i : arr) {**

// 这个地方可以等于0或者1，因为

// eor & (取反eor+1) = 0000000100，如果(i & rightOne) == 0,则证明取出来的该位置的为0，例如 \*\*\*\*\*\*\*000，

// 如果(i & rightOne) == 1,则证明取出来的该位置的为1，例如 \*\*\*\*\*\*\*100，由于 是奇数位，所以

// 如果(i & rightOne) == 0,则将该值^运算,假如他是a则，a^a^a = a,此时onlyOne的值为a

// 如果(i & rightOne) == 1,则将该值^运算,假如他是b则，b^b^b = b,此时onlyOne的值为b

// 因为此处任取一边即可，即可以=0或者=1

**if ((i & rightOne) == 0) {**

**onlyOne ^= i;**

**}**

**}**

**System.err.println(onlyOne + " " + (eor ^ onlyOne));**

**}**

### $6\_插入排序

时间复杂度O(n^2),额外空间复杂度O(1)。插入排序相对比选择，冒泡好些 将新数据插入已经排好序的列中，插入排序的 时间复杂度与数据情况有关，例如 7654321 6754321 处理6的数据，向前<7跟7交换。换1次。 5674321 处理5的数据，向前<7跟7交换，继续向前判断，<6.跟 6交换，换2次。 4567321 处理4的数据，向前<7跟7交换，继续向前判断，<6.跟 6交换，继续向前判断，<5.跟 5交换，换3次。 ... 此时为等差数列，**时间复杂度为O(n^2) 如果是1234567 只需要判断一次，此时时间复杂度为O(n)**

**public static void process1(int[] arr) {**

**if (arr == null || arr.length <= 1) {**

**return;**

**}**

**// 遍历第二个到最后一个**

**for (int i = 1; i < arr.length; i++) {**

**// 与前一个元素比较，如果前一个元素大于当前元素，则交换位置**

**for (int j = i - 1; j >= 0 && arr[j] > arr[j + 1]; j--) {**

**swap(arr, j, j + 1);**

**}**

**}**

**}**

**private static void swap(int[] arr, int i, int j) {**

**if (j != i) {**

**// 前提条件：i和j不能用同一块内存区域**

**arr[i] = arr[i] ^ arr[j];**

**arr[j] = arr[i] ^ arr[j];**

**arr[i] = arr[i] ^ arr[j];**

**}**

**}**

### $7\_二分法找有序数组中的某个数字

时间复杂度O(logN） 每次 砍一半 8 4 2 1，找区间中某个数，如果存在，返回下标，不存在，返回-1。 如果8个数，**需要砍的最多的次数为 4 2 1，共3次。2^3 = 8，所以时间复杂度为O(logN)**

**public static int binarySearch(int[] nums, int target) {**

**if (nums == null || nums.length == 0) {**

**throw new IllegalArgumentException("输入数组为空或null");**

**}**

**int left = 0;**

**int right = nums.length - 1;**

**while (left <= right) {**

**int mid = left + (right - left) / 2;**

**// 检查中间值是否等于目标值**

**if (nums[mid] == target) {**

**return mid; // 找到目标值，返回其索引**

**}**

**// 如果中间值小于目标值，说明目标值在右半部分**

**if (nums[mid] < target) {**

**left = mid + 1;**

**} else {**

**// 如果中间值大于目标值，说明目标值在左半部分**

**right = mid - 1;**

**}**

**}**

**// 如果循环结束仍未找到目标值，返回 -1**

**return -1;**

**}**

**public static void main(String[] args) {**

**int[] nums = {1, 2, 3, 4, 5, 6, 7, 8, 9};**

**int target = 5;**

**int index = binarySearch(nums, target);**

**if (index != -1) {**

**System.out.println("目标值 " + target + " 在数组中的索引为: " + index);**

**} else {**

**System.out.println("目标值 " + target + " 不在数组中");**

**}**

**}**

### $8\_二分法找有序数组中大于等于某个数最左侧的位置

**时间复杂度O(logN）**

**public static int findLeftmostGreaterOrEqual(int[] nums, int target) {**

**if (nums == null || nums.length == 0) {**

**throw new IllegalArgumentException("输入数组为空或null");**

**}**

**int left = 0;**

**int right = nums.length - 1;**

**int result = -1; // 用于记录大于等于目标值的最左侧位置**

**while (left <= right) {**

**int mid = left + (right - left) / 2;**

**// 检查中间值是否大于等于目标值**

**if (nums[mid] >= target) {**

**result = mid; // 更新结果为当前中间位置**

**right = mid - 1; // 继续在左半部分查找更左侧的位置**

**} else {**

**left = mid + 1; // 目标值在右半部分**

**}**

**}**

**return result;**

**}**

### $9\_局部最小

无序数组中 ，相邻的数一定不相等， **时间复杂度：O(logN)**，因为每次迭代将搜索范围减半。 空间复杂度：O(1)，只使用了常数级的额外空间。

i 输入检查：首先检查输入数组是否为空或长度为0，如果是则抛出异常。 初始化左右指针：left 初始化为0，right 初始化为数组长度减1。 二分查找： 计算中间索引 mid。 检查 mid 是否为局部最小值： 如果 mid 是第一个元素且小于其右邻居，或者 mid 是最后一个元素且小于其左邻居，或者 mid 小于其左右邻居，则 mid 是局部最小值，返回 mid。 如果 mid 的左邻居小于 mid，则局部最小值在左半部分，移动 right 到 mid - 1。 否则，局部最小值在右半部分，移动 left 到 mid + 1。 异常处理：如果循环结束仍未找到局部最小值，抛出异常。理论上，只要输入数组有效且相邻元素不相等，这种情况不会发生。

**public static int findLocalMin(int[] nums) {**

**if (nums == null || nums.length == 0) {**

**throw new IllegalArgumentException("Input array is empty or null");**

**}**

**int left = 0;**

**int right = nums.length - 1;**

**while (left <= right) {**

**int mid = left + (right - left) / 2;**

// 检查 mid 是否为局部最小值

// (mid == 0 || nums[mid] < nums[mid - 1])检查 mid 是否为数组的第一个元素或小于其左邻居：

//(mid == nums.length - 1 || nums[mid] < nums[mid + 1])检查 mid 是否为数组的最后一个元素或小于其右邻居

**if ((mid == 0 || nums[mid] < nums[mid - 1]) && (mid == nums.length - 1 || nums[mid] < nums[mid + 1])) {**

**return mid; // Return the index of the local minimum**

**}**

//如果 mid 的左邻居小于 mid，则局部最小值在左半部分

// 1,3,2,3,4

**if (mid > 0 && nums[mid - 1] < nums[mid]) {**

**right = mid - 1;**

**} else {**

**// 否则，局部最小值在右半部分**

**left = mid + 1;**

**}**

}

// T这种情况理论上不会发生，因为输入数组有效且相邻元素不相等

**throw new IllegalStateException("未找到局部最小值");**

**}**

### $10\_对数器01

**可以对比 数算法是否正确，使用常规写法得到的数据和自己实现的算法得到的结果数据，进行每一个的比对 ，指定随机数组，随机对比次数，来比较**

**public static int[] generateRandomArray(int maxSize, int maxValue) {**

**// 数组长度随机，1~maxSize**

**int[] arr = new int[(int) ((maxSize + 1) \* Math.random())];**

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

**arr[i] = (int) ((maxValue + 1) \* Math.random()) - (int) (maxValue \* Math.random());**

**}**

**return arr;**

**}---------------------------------------------------------------------------------------------**

**public static boolean isEqual(int[] arr1, int[] arr2) {**

**if ((arr1 == null && arr2 != null) || (arr1 != null && arr2 == null)) {**

**return false;**

**}**

**if (arr1 == null && arr2 == null) {**

**return true;**

**}**

**if (arr1.length != arr2.length) {**

**return false;**

**}**

**for (int i = 0; i < arr1.length; i++) {**

**if (arr1[i] != arr2[i]) {**

**return false;**

**}**

**}**

**return true;**

**}---------------------------------------------------------------------------------------------**

**public static int[] copyArray(int[] arr) {**

**if (arr == null) {**

**return null;**

**}**

**int[] res = new int[arr.length];**

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

**res[i] = arr[i];**

**}**

**return res;**

**}---------------------------------------------------------------------------------------------**

### $10\_对数器02

**public static void main(String[] args) {**

**// int[] arr = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 99, 0, 33, 5};**

**// process1(arr);**

**// System.err.println(Arrays.toString(arr));**

**int testTime = 500000;**

**int maxSize = 100;**

**int maxValue = 100;**

**boolean success = true;**

**for (int i = 0; i < testTime; i++) {**

**int[] arr1 = generateRandomArray(maxSize, maxValue);**

**int[] arr2 = copyArray(arr1);**

**process1(arr1);**

**Arrays.sort(arr2);**

**System.err.println(i + "========" + Arrays.toString(arr1));**

**System.err.println(i + "========" + Arrays.toString(arr2));**

**if (!isEqual(arr1, arr2)) {**

**success = false;**

**break;**

**}**

**}**

**System.err.println(success ? "Nice!" : "Fucking fucked!");**

**}**

**private static void swap(int[] arr, int i, int j) {**

**if (j != i) {**

**// 前提条件：i和j不能用同一块内存区域**

**arr[i] = arr[i] ^ arr[j];**

**arr[j] = arr[i] ^ arr[j];**

**arr[i] = arr[i] ^ arr[j];**

**}**

**}**

### $11\_递归LR范围取最大值-理论

\* 递归结构图如下

\* [3,2,5,6,7,4]

\* 0,1,2,3,4,5

\* P(arr,0,5):

\* P(0,2)

\* P(0,1)

\* P(0,0)

\* P(1,1)

\* P(2,2)

\* P(3,5)

\* P(3,4)

\* P(3,3)

\* P(4,4)

\* P(5,5)

\* 由上可知，递归结束条件为L=R

\* 使用master公式估算时间复杂度：

\* · T(n) = 2T(n/2)+O(1)

\* 第一个2指的调用了两次：

\* int leftMax = max(arr, L, mid);

\* int rightMax = max(arr, mid + 1, R);

\* n/2指的是里面的数量分别为n/2

\* max(arr, L, mid);

\* max(arr, mid + 1, R);

\* 第三个O(1)指的是：

\* int mid = L + ((R - L) >> 1);

\* return Math.max(leftMax, rightMax);

\* 如果将LR之间的区域划分为均等三份分别求最大值，则时间复杂度公式为：

\* T(n) = 3T(n/3)+O(1) ，3表示3次 n/3表示数据规模，也符合master公式

\* 如果将LR直接的区域分为两份，左边1/3，右边2/3，则时间复杂度为：

\* T(n) = T(n/3)+T(2n/3)+O(1)，此时就不符合master公式

**public static int max(int[] arr, int L, int R) {**

**if (L == R) {**

**return arr[L];**

**}**

**int mid = L + ((R - L) >> 1);//二分中位数**

**int leftMax = max(arr, L, mid);//左最大**

**int rightMax = max(arr, mid + 1, R);//右最大**

**return Math.max(leftMax, rightMax);//求最大**

**}**

**int[] arr = {1, 2, 3, 4, 5, 6, 7, 8, 9};**

**System.out.println(max(arr, 0, arr.length - 1));**

### $12\_递归LR范围归并排序

使用master公式计算时间复杂度： T(n) = 2T(n/2) + mergeV2的时间复杂度 左侧：left->Mid 右侧：Mid->right 最后copy到原数组：所以复杂度为 O(N) 所以最后为 T(n) = 2T(n/2) + O(N) 时间复杂度为：O(NlogN)，额外空间复杂度为O(N)，用来merge，每次都是用完即释放，最多的空间就是长度为N的空间 之前说的选择，冒泡，插入都是O(N^2)，为什么会比那个好？？？ 因为那三个大量浪费了比较行为才搞定了一个数，而归并排序，是把左右两侧的数排序好，然后合并，相比 而言是没有浪费比较行为的 实际可以理解为：**切分为有序的小块，利用辅助空间，比较每个小块每个位置的数据，进行合并，所以其实并没有浪费比较行为**

**public static void process(int[] arr, int L, int R) {**

**if (L == R) {**

**return;**

**}**

**int mid = L + ((R - L) >> 1);**//二分中位数

**process(arr, L, mid)**;//左侧有序

**process(arr, mid + 1, R);**//右侧有序

**mergeV2(arr, L, mid, R);**//marge左右两侧

**}**

**private static void mergeV2(int[] arr, int left, int mid, int right) {**

**int[] help = new int[right - left + 1]; /**/开辟辅助空间，存储左右两侧排序后的数据

**int i = 0;** //专门给help添值的下标

**int p1 = left;** //左侧开始位置

**int p2 = mid + 1;**  //右侧开始位置

**while (p1 <= mid && p2 <= right) {**//p1不越界的情况下，并且p2也不越界的情况下

**if (arr[p1] <= arr[p2]) {**

**help[i++] = arr[p1++];**

**} else {**

**help[i++] = arr[p2++];**

**}**//以上写法等价于：help[i++] = arr[p1] <= arr[p2] ? arr[p1++] : arr[p2++];

**}**

**while (p1 <= mid) {**//如果P1没越界，P2越界了，就将P1剩下的数据拷贝到help

**help[i++] = arr[p1++];**

**}**

**while (p2 <= right) {**//如果P2没越界，P1越界了，就将P2剩下的数据拷贝到help

**help[i++] = arr[p2++];**

**}**

**for (int j = 0; j < help.length; j++) {**//将临时辅助空间的数据复制到原数组中

**arr[left + j] = help[j];**

**}**

**}**

### $13\_递归小和问题

在一个数组中，每一个数左边比当前数小的数累加起来， 叫做这个数组的小和。求一个数组的小和。 例子:[1,3,4,2,5]1左边比1小的数，没有; 3左边比3小的数，1; 4左边比4小的数，1、3; 2左边比2小的数，1; 5左边比5小的数，1、3、4、2; 所以小和为1+1+3+1+1+3+4+2=16 ------------------------------- **换思路： [1,3,4,2,5] 第一个：求右边有几个数比1大：1\*4 第二个：求右边有几个数比3大：3\*2 第三个：求右边有几个数比4大：4\*1 第四个：求右边有几个数比2大：2\*1 加和 = 16**

**public static int smallSum(int[] arr) {**

**if (arr == null || arr.length < 2) {return 0;}**

**return processV2(arr, 0, arr.length - 1);**

**}-----------------------------------------------------------------------**

**private static int processV2(int[] arr, int l, int r) {**

**if (l == r) {return 0;}**

**int mid = l + ((r - l) >> 1);**

// 返回左侧排序求小和数量 + 右侧排序求小和数量 + 左右合并merge求小和数量

**return processV2(arr, l, mid) + processV2(arr, mid + 1, r) + mergeV3(arr, l, mid, r);**

**}-----------------------------------------------------------------------**

**private static int mergeV3(int[] arr, int l, int mid, int r) {**

**// [1,3,4,2,5] l=0 mid=2 r=5**

**int[] help = new int[r - l + 1];**

**int i = 0;**

**int p1 = l;**

**int p2 = mid + 1;**

**int res = 0;**

**while (p1 <= mid && p2 <= r) {//都没有下标越界的时候**

**if (arr[p1] < arr[p2]) {//只有左组比右组小，才算小和**

**res += (r - p2 + 1) \* arr[p1];//计算方式为 个数\* 当前数（即小的这个数）**

**help[i++] = arr[p1++];**

**} else {//如果大于或者等于，先移动右边的数，不满足条件不计算**

**help[i++] = arr[p2++];**

**}**

**}**

**while (p1 <= mid) {**

**help[i++] = arr[p1++];**

**}**

**while (p2 <= r) {**

**help[i++] = arr[p2++];**

**}**

**for (int j = 0; j < help.length; j++) {**

**arr[l + j] = help[j];**

**}**

**return res;}**

### $14\_递归逆序对问题

逆序对问题 在一个数组中，左边的数如果比右边的数大，则这两个数构成一个逆序对，请打印所有逆序对。或者找到 逆序对个数 32450 求右边的数有多少个比左边的数小 3:2个 2:1个 4:1个 5:1个 总计 5个

private int[] temp;

public int reversePairs(int[] nums) {

temp = new int[nums.length];

return mergeSort(nums, 0, nums.length - 1);

}

private int mergeSort(int[] nums, int left, int right) {

if (left >= right) {return 0;}

int mid = left + (right - left) / 2;

int count = mergeSort(nums, left, mid) + mergeSort(nums, mid + 1, right);

count += merge(nums, left, mid, right);

return count;

}

private int merge(int[] nums, int left, int mid, int right) {

for (int i = left; i <= right; i++) {

temp[i] = nums[i];

}

int i = left, j = mid + 1;

int count = 0;

int k = left;

while (i <= mid && j <= right) {

if (temp[i] <= temp[j]) {

nums[k++] = temp[i++];

} else {

nums[k++] = temp[j++];

count += (mid - i + 1); // 统计逆序对

}

}

while (i <= mid) {

nums[k++] = temp[i++];

}

while (j <= right) {

nums[k++] = temp[j++];

}

return count;

}

int[] nums = {3, 2, 4, 5, 0};

int count = solution.reversePairs(nums);System.out.println("逆序对总数: " + count);