1 认识复杂度和简单排序算法

1.1 常数操作

与数据量的大小无关

int a = arr[i];　计算一个偏移量即可，属于常数操作

list.get(i);　链表需要一个一个遍历，一个一个跳，不是常数操作

加减乘除是常数操作

1.2 选择排序

1.2.1 原理

0~N-1，找到最小值的索引，值放到 0 位置上；

1~N-1，找到最小值的索引，值放到 1 位置上；

……

1.2.2 时间复杂度（常数操作数量的指标）

看：N + (N-1) + (N-2)+……

比：N + (N-1) + (N-2)+……

换：N

常数操作次数 $= aN^2 + bN + c$，不要低阶项，不要系数

$O(N^2)$

1.2.3 评价算法好坏

先看时间复杂度，当时间复杂度相同时，再分析实际运行时间（如加减乘除与位运算或与异或），以此比较出常数项时间

1.2.4 代码

```
void selectSort(int* arr, int len)
{
	if (arr == NULL || len < 2)
	{
		return;
	}
	for (int i = 0; i < len-1; i++)	//i~N-1
	{
		int minIndex = i; //记录最小值的索引
		for (int j = i + 1; j < len; j++)  //i~N-1
		{
			minIndex = arr[j] < arr[minIndex] ? j : minIndex;
		}
		swap(arr, i, minIndex);
	}
}
```

1.2.5 额外空间复杂度

额外空间复杂度 O(1)——只需要几个额外变量

若是需要额外数组，且大小与原数组相同，则为 O(N)

1.3 冒泡排序

1.3.1 原理

每次都从 0 位置开始，相邻谁大谁右移，第一轮搞定 N-1 位置上的数，第二轮搞定 N-2 位置上的数

1.3.2 时间复杂度

$O(N^2)$

1.3.3 额外空间复杂度

O(1)

1.3.4 代码

```
void bubbleSort(int* arr, int len)
{
	if (arr == NULL || len < 2)
	{
		return;
	}
	for (int end = len - 1; end > 0; end--)    //每次搞定最后一个数, N-1, N-2,
	{
		for (int j = 0; j < end; j++)//0~end
		{
			if (arr[j] > arr[j + 1])  //相邻谁大谁右移
			{
				swap(arr, j, j + 1);
			}
		}
	}
}
```

1.4 异或^

1.4.1 基本原理

两个数异或，相同为 0 不同为 1。或者理解为无进位相加。

(1) 0 ^ N = N

(2) N ^ N = 0

(3) 满足交换律和结合律

(4) 一堆数异或，只跟数量有关，与顺序无关

1.4.2 swap（数组中前提：i、j 是内存的两块不同位置）

```
int a = 甲;
```

```
int b = 乙;
a = a ^ b;    a = 甲^乙; b = 乙;
b = a ^ b;    a = 甲^乙; b = 甲^乙^乙 = 甲;
a = a ^ b;    a = 甲^乙^甲 = 乙; b = 甲;
```

1.5 异或面试题

1.5.1 问题描述

一个数组内全是 int，只有一种数出现了奇数次，其他出现了偶数次

(1) 如何找到出现奇数次的数？

(2) 如果两种数出现了奇数次，其他偶数次，如何找出这两个数？

1.5.2 第一问解

准备一个 eor = 0，把数组内的每一个数都异或一遍，得到的结果即为出现奇数次的数。

```
void printOddTimesNum1(int* arr, int len)
{
    int eor = 0;
    for (int i = 0; i < len; i++)
    {
        eor ^= arr[i];
    }
    cout << eor << endl;
}
```

注：无进位相加。每一位的值由所有数该位上的 1 的个数决定。

1.5.3 第二问解

先准备一个 eor，把数组所有数都异或一遍，最终 eor = a ^ b

由于 a≠b，那么 eor≠0，说明 eor 的某一位不等于 0，即 eor 的某一位是 1

假设第八位是 1，说明 a、b 的第八位不同，一个是 1 一个是 0

准备一个新的 eor’，把所有第八位是 1 的数异或一遍，就可以得到 a 或者 b

再用 eor’异或 eor，得到另一个数

出现偶数次的 other1 a 和 b 不在同一个集合内，假设 a 在这 a eor’异或第八位是 1 的所有数 eor’ = a； eor’ ^ eor = b； 第八位是 1	出现偶数次的 other2 b 第八位是 0

找到最右侧的 1， eor & (~eor + 1)

eor 1010111100

~eor 0101000011

~eor + 1 0101000100

eor & (~eor + 1) 0000000100

找这一位上全是 1 的数，arr[i] & onlyOne == 1

一个数“与”上 0000000100，想得到 1，那么这个数的右边第三位上一定是 1，这样通过遍历就找出了所有这一位上是 1 的数。

```
void printOddTimesNum2(int* arr, int len)
{
	int eor = 0;
	for (int i = 0; i < len; i++)
	{
		eor ^= arr[i];
	}
	//此时eor = a^b
	//eor != 0
	//eor必有一个位置上是1
	//我们选最右侧的1
	int rightOne = eor & (~eor + 1);

	int onlyOne = 0; //eor'
	for (int i = 0; i < len; i++)
	{
		if ((arr[i] & rightOne) == 1)    //这一位是1的数
		{
```

```
            onlyOne ^= arr[i];      //才^
        }
    }
    cout << onlyOne << " " << (eor ^ onlyOne) << endl;
}
```

1.6 插入排序

1.6.1 原理（斗地主抓牌）

0~0 有序

0~1，拿索引为 1 的数往前看，比左小换，一直比下去，直到不换为止

0~2，拿索引为 2 的数往前看……

第 i 轮拿第 i 个数往左比

1.6.2 时间复杂度

数据状况不同，**时间复杂度**不同，**最差交换次数 O(N^2)**，最好 O(N)

1.6.3 额外空间复杂度

O(1)

1.6.4 代码

```
void insertionSort(int* arr, int len)
{
    if (arr == NULL || len < 2)
    {
        return;
    }
    //0~0有序，0~i想有序
    for (int i = 1; i < len; i++)
    {
        for (int j = i - 1; j >= 0 && arr[j] > arr[j + 1]; j--)      //j是当前数的前一个位
置
        {
```

```
                swap(arr, j, j + 1);
            }
        }
}
```

1.7 二分法

1.7.1 在一个有序数组中，找某个数是否存在

遍历 O(N)

找 mid 值，若大于 num，右边的数不要了；若小于 num，左边的数不要了

$O(\log_2 N)$，写作 O(logN)

1.7.2 在一个有序数组中，找>=某个数最左侧的位置

假设找>=3 的最左侧的位置。二分，mid 满足，用 t 记录索引，往左找；二分，mid 不满足，往右找；二分，mid 满足，是否比 t 索引更小，更新 t；……；直到二分到结束为止。

1.7.3 局部最小值问题

arr 无序，任意相邻数一定不相等，求一个局部最小的位置，好于 O(N)。

局部最小：arr[0] < arr[1]，0 位置是局部最小；arr[N-1] < arr[N-2]，N-1 位置是局部最小；arr[i-1] < arr[i] <arr[i+1]，i 位置是局部最小。

先判断 base case，0 位置和 N-1 位置，若未返回：左侧单调递减，右侧单调递增，则内部必存在一个拐点，也就是局部最小点。二分，mid 若满足要求则返回，不满足，假设 mid-1<mid，那么左侧必存在一个拐点，即局部最小点。

1.8 对数器

方法 a：想测的方法

方法 b：很好写，一定对，不考虑时间复杂度的方法

随机样本产生器，产生样本分别给两个方法得到 ret1 和 ret2，ret1==ret2 则对；若不相等，先产生少量样本，把每个方法调整好，再增大样本数量。

1.9 递归求最大值

mid = (L+R)/2，可能溢出，mid = L + (R-L)/2 = L+((R-L)>>1)

1.9.1 代码

```
int process(int* arr, int L, int R)
{
	if (L == R)
	{
		return arr[L];
	}
	int mid = L + ((R - L) >> 1);
	int leftMax = process(arr, L, mid);
	int rightMax = process(arr, mid + 1, R);
	return max(leftMax, rightMax);
}
```

利用栈玩了一个遍历（树）

a = 2, b = 2, d = 0　　　　O(N)

1.9.2 master 公式

$$T(N) = a * T(\frac{N}{b}) + O(N^d)$$

T(N)：母问题数据量

a：子问题调用次数

N/b：子过程规模等量

$O(N^d)$：除了子问题的调用外，剩下的 bigO

时间复杂度

$$log_b a < d, O(N^d)$$

$$log_b a > d, O(N^{log_b a})$$

$$log_b a == d, O(N^d * logN)$$

2 认识 O(NlogN)的排序

2.1 归并排序

2.1.1 原理

从 L 到 R 分两半，M，**分到数组的最小单位**。先将左侧有序，再将右侧有序，再 merge 整合。整合时，准备两个指针一个辅助数组，哪个指针的数小就放到辅助数组里，然后移动指针和辅助数组的下标。

先使用递归，找到最底下的两个数，merge，再依次传上去 merge。

外排序。

2.1.2 时间复杂度

a = 2，b = 2，d = 1；logab = d；O(NlogN)

2.1.3 额外空间复杂的

O(N)

2.1.4 代码

```
class MergeSort
{
public:
    void mergeSort(int* arr, int len)
    {
        if (arr == NULL || len < 2)
        {
            return;
        }
        process(arr, 0, len - 1);
    }

    void process(int* arr, int L, int R)
    {
        if (L == R)
        {
```

```
        return;
    }
    int mid = L + ((R - L) >> 1);
    process(arr, L, mid);
    process(arr, mid + 1, R);
    merge(arr, L, mid, R);
}

void merge(int* arr, int L, int M, int R)
{
    int* temp = new int[R - L + 1];
    for (int j = 0; j < R - L + 1; j++)
    {
        temp[j] = 0;
    }
    int i = 0;
    int p1 = L;
    int p2 = M + 1;
    while (p1 <= M && p2 <= R)
    {
        temp[i++] = arr[p1] <= arr[p2] ? arr[p1++] : arr[p2++];
    }
    while (p1 <= M)
    {
        temp[i++] = arr[p1++];
    }
    while (p2 <= R)
    {
        temp[i++] = arr[p2++];
    }
    for (i = 0; i < R-L+1; i++)
    {
        arr[L + i] = temp[i];
    }
}
};
```

2.1.5 优点

比较行为没有被浪费，变成了整体有序的部分，往下传递进行 merge。

2.2 小和问题

2.2.1 问题描述

有一个数组，第 i 个位置左边比它小的数的求和，把所有的小和加起来。

2.2.2 分析

左边比它小的等价于，第 i 个位置右边有多少个比它大的数，它对小和的贡献为个数×arr[i]。左侧小时才产生小和，并且发生左侧 copy。

利用 merge，在左严格小于右时，res+=左值*(r-p2+1)。

注意，当左右相等时，先 copy 右，边排序边计算小和。

2.2.3 代码

```
class SmallSum
{
public:
    int smallSum(int* arr, int len)
    {
        if (arr == NULL || len < 2)
        {
            return 0;
        }
        return process(arr, 0, len - 1);
    }

    int process(int* arr, int l, int r)
    {
        if (l == r)
        {
            return 0;
        }
        int mid = l + ((r - l) >> 1);
        return process(arr, l, mid) + process(arr, mid + 1, r) + merge(arr, l, mid, r);
    }

    int merge(int* arr, int L, int m, int r)
    {
        int* temp = new int[r - L + 1];
```

```
        for (int j = 0; j < r - L + 1; j++)
        {
            temp[j] = 0;
        }
        int i = 0;
        int p1 = L;
        int p2 = m + 1;
        int res = 0;
        while (p1 <= m && p2 <= r)
        {
            res += arr[p1] < arr[p2] ? (r - p2 + 1) * arr[p1] : 0;
            temp[i++] = arr[p1] < arr[p2] ? arr[p1++] : arr[p2++];
        }
        while (p1 <= m)
        {
            temp[i++] = arr[p1++];
        }
        while (p2 <= r)
        {
            temp[i++] = arr[p2++];
        }
        for (i = 0; i < r - L + 1; i++)
        {
            arr[L + i] = temp[i];
        }
        return res;
    }
};
```

2.3 逆序对问题

2.3.1 问题描述

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。

2.3.2 分析

右边比左边小，跟 merge 一样。

2.3.4 代码（部分）

```
while (p1 <= m && p2 <= r)
{
    res += arr[p1] > arr[p2] ? (r - p2 + 1) : 0;
    temp[i++] = arr[p1] > arr[p2] ? arr[p1++] : arr[p2++];
}
```

2.4 快速排序（荷兰国旗问题）

2.4.1 问题描述

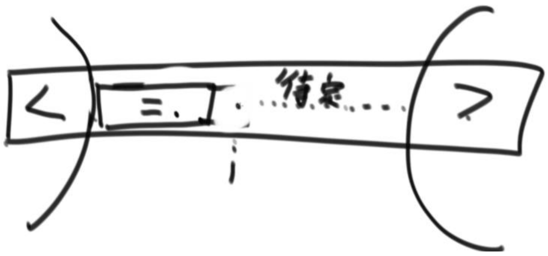
(1) 给定一个数组 arr，和一个数 num，请把小于等于 num 的数放在数组的左边，大于 num 的数放在数组的右边。要求额外空间复杂度 O(1)，时间复杂度 O(N)

(2) 小于放左边，等于放中间，大于放右边

2.4.2 分析

(1) 准备一个小于等于区域的右边界，和 i。[i]≤num，[i]和≤区域的下一个数交换，≤区右扩，i++；[i]>num，i++；直到 i 越界。

(2) 准备一个小于区的右边界和大于区的左边界，分别位于两端，和 i。[i]<num，[i]和<区域的下一个数交换，<区右扩，i++；[i]=num，i++；[i]>num，[i]和>区域的前一个数交换，i 原地不动（因为换过来的数是新来的，没见过）；直到>区与 i 撞上了停。（压缩待定区域）



2.4.3 快排 1.0

一个数组，拿最后一个数当作 num，小于等于放左边，大于放右边。分好后，把 num 和大于区的第一个数交换（这样这个数就处在正确位置上了，固定）。接下来对左右两个区域递归，做同样的操作。$O(N^2)$

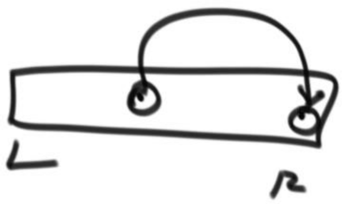
2.4.4 快排 2.0

拿最后一个数，把数组分为小于区，等于区，大于区。把 num 和大于区第一个数交换，**一次搞定一批数**，再在左右两区域递归。$O(N^2)$

2.4.5 快排 3.0

当前两个版本，划分值选的很偏时，会出现差情况。好情况是划分值打到中间，master 公式 a=2，b=2，d=1，O(NlogN)。越偏越会趋近于 $O(N^2)$。

随机选一个数，把它放到最后一个位置上，再进行划分操作。



2.4.6 代码

```
class QuickSort
{
public:
	void quickSort(int* arr, int len)
	{
		if (arr == NULL || len < 2)
		{
			return;
		}
		process(arr, 0, len - 1);
	}

	void process(int* arr, int L, int R)
	{
		if (L < R)
		{
			//swap
			int* p = partition(arr, L, R);
			process(arr, L, p[0] - 1);
			process(arr, p[1] + 1, R);
		}
	}
	//处理arr[l..r]的函数
	//默认以arr[r]做划分，arr[r]->p <p ==p >p
	//返回等于区域（左边界，右边界），所以返回一个长度为2的数组res，
res[0] res[1]
	int* partition(int* arr, int L, int R)
	{
		int less = L - 1;   //<区右边界（-1）
		int more = R;       //>区左边界（R，因为划分值是arr[R]，所以R位置是大
于区左边界）
		while (L < more)//L表示当前数的位置 arr[R]->划分值（当前数未撞到大
于区边界）
		{
			/*
			less [L                                |R(==more)]
			后面less和more都是该区域最边界的索引，符合该区域要求，即less
是小于区最后一个数，more是大于区第一个数
			*/
			if (arr[L] < arr[R])     //当前数 < 划分值
			{
				swap(arr, ++less, L++);    //交换小于区的下一个数++less和当前
```

```
数L++
			}
			else if (arr[L] > arr[R])	//当前数 > 划分值
			{
				swap(arr, --more, L); //交换大于区的前一个数mroe--和当前数
L，L原地不动，因为换过来的数是新的
			}
			else
			{
				L++;	//等于，L++
			}
		}
		swap(arr, more, R);	//交换大于区的第一个数和最后一个数R
		int p[] = { less + 1,more };	//返回等于区域，less+1和more（==之前的
R）
		return p;
	}

	void swap(int* arr, int i, int j)
	{
		int temp = arr[i];
		arr[i] = arr[j];
		arr[j] = temp;
	}
};
```

3 详解桶排序以及排序内容大总结