啥叫时间复杂度啊？啥叫空间复杂度啊？

刚开始学编程、刷算法题的小伙伴，大概率会被“时间复杂度”“空间复杂度”这两个词搞懵——明明代码能正常运行，为啥还要纠结这两个东西？其实，这两个概念是衡量算法“好坏”的核心标准：时间复杂度看算法“跑得多快”，空间复杂度看算法“占多少内存”。不同于具体代码的运行耗时、内存占用（受设备、数据量影响），复杂度是**脱离具体环境的通用评价指标**，能帮我们快速判断算法的效率和资源消耗，也是面试、算法选型的关键依据。本文将用最通俗的语言+实战案例，把两个概念讲透，彻底摆脱困惑。

一、先搞懂核心：复杂度到底在衡量什么？

不管是时间复杂度还是空间复杂度，核心目的都是“量化算法的资源消耗”，帮我们在不运行代码的情况下，快速对比不同算法的优劣。这里要明确一个关键前提：复杂度关注的是“数据量变化时，资源消耗的变化趋势”，而非具体的消耗数值——比如同样是排序，数据量从10个增加到10000个，有的算法耗时翻倍，有的算法耗时暴涨，这就是复杂度要描述的“趋势”。

举个生活化的例子：煮一碗面和煮100碗面，两种做法的“复杂度”不同：煮1碗面，不管多少人吃，只要煮一锅就够（资源消耗不随人数增加而明显变化）；煮100碗面，要么分100锅煮（耗时随人数线性增加），要么用更大的锅煮（占用更多空间）。这就是“时间”和“空间”消耗的不同趋势，对应到算法中，就是时间复杂度和空间复杂度。

二、时间复杂度：算法“跑得多快”的衡量标准

2.1 核心定义

时间复杂度（Time Complexity），指的是**算法执行所需的时间，随数据量n变化的趋势**，通常用大O符号（O()）表示，比如O(n)、O(logn)、O(n^2)。

注意：时间复杂度不是“具体运行时间”——同一算法在不同设备（电脑、手机）上运行时间不同，但它的时间复杂度是固定的。我们关注的是“数据量n增大时，算法耗时增长的快慢”：增长越慢，算法效率越高。

2.2 怎么判断时间复杂度？（核心方法）

判断时间复杂度，无需逐行统计代码执行次数，只需抓住“最耗时的核心操作”，遵循3个简化原则即可：

1. 只关注“循环/递归次数最多”的代码块：算法的耗时主要集中在重复执行的操作上，单次执行的代码（如变量定义、赋值）可忽略；
2. 忽略常数项和低次项：比如执行次数是3n+5，简化为O(n)；执行次数是n^2+2n+3，简化为O(n^2)——因为当n足够大时，低次项和常数项对趋势的影响可以忽略；

3. 忽略系数：比如执行次数是2n，简化为O(n)；执行次数是$3n^2$，简化为$O(n^2)$——系数只影响具体耗时，不影响“增长趋势”。

2.3 常见时间复杂度及案例（从优到劣排序）

不同时间复杂度的效率差异极大，尤其是数据量较大时（n>10000），差距会被无限放大，以下是最常见的几种：

（1）常数复杂度 O(1)：效率最高

核心特点：算法执行时间不随数据量n变化，始终是固定值，不管n是10还是10000，耗时都一样。常见于无循环、无递归的简单操作。

案例：获取数组的第i个元素、两数相加、变量赋值。

```cpp
#include <iostream>
#include <vector>
using namespace std;

int main() {
    vector<int> arr = {1, 2, 3, 4, 5};
    int i = 2;
    // 核心操作：获取数组第i个元素，仅执行1次，与n无关
    int val = arr[i];
    cout << val << endl; // 输出3
    return 0;
}

```

（2）对数复杂度 O(logn)：效率极高

核心特点：算法执行时间随n增长，但增长速度极慢，n越大，优势越明显（比如n=100万时，$\log_2 n$仅约20）。常见于“每次操作都将数据量减半”的场景，最典型的是二分查找。

案例：二分查找（在有序数组中找目标元素）。

```cpp
#include <iostream>
#include <vector>
using namespace std;

// 二分查找：有序数组arr中找target，时间复杂度O(logn)
int binarySearch(vector<int>& arr, int target) {
    int left = 0, right = arr.size() - 1;
    // 每次循环，查找范围减半（right-left缩小一半）
    while (left <= right) {

```

```
        int mid = (left + right) / 2;
        if (arr[mid] == target) return mid;
        else if (arr[mid] > target) right = mid - 1;
        else left = mid + 1;
    }
    return -1; // 未找到
}

int main() {
    vector<int> arr = {1, 3, 5, 7, 9, 11};
    cout << binarySearch(arr, 7) << endl; // 输出3
    return 0;
}

```

（3）线性复杂度 O(n)：效率中等

核心特点：算法执行时间与数据量n成正比，n翻倍，耗时也翻倍。常见于单循环操作（循环次数与n一致）。

案例：线性查找、遍历数组、累加求和。

```
#include <iostream>
#include <vector>
using namespace std;

// 累加求和：时间复杂度O(n)，循环n次
int sum(vector<int>& arr) {
    int total = 0;
    for (int num : arr) { // 循环次数等于数组长度n
        total += num;
    }
    return total;
}

int main() {
    vector<int> arr = {1, 2, 3, 4, 5};
    cout << sum(arr) << endl; // 输出15
    return 0;
}

```

（4）线性对数复杂度 O(nlogn)：效率较好

核心特点：执行时间 = 线性操作（O(n)） + 对数操作（O(logn)），n翻倍时，耗时约翻倍再乘以$\log_2 2$（即翻倍），效率优于$O(n^2)$，是主流排序算法的常用复杂度。

案例：快速排序、归并排序、堆排序（C++的sort函数底层就是O(nlogn)的混合排序）。

（5）平方复杂度 $O(n^2)$：效率较低

核心特点：执行时间与n的平方成正比，n翻倍，耗时翻4倍；n=1000时，执行次数达100万次，仅适用于小数据量场景。常见于双重嵌套循环。

案例：冒泡排序、选择排序、双重循环遍历。

```cpp
#include <iostream>
#include <vector>
using namespace std;

// 冒泡排序：时间复杂度O(n²)，双重嵌套循环
void bubbleSort(vector<int>& arr) {
    int n = arr.size();
    // 外层循环n次，内层循环n-i-1次，总执行次数约n²/2
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j+1]) {
                swap(arr[j], arr[j+1]);
            }
        }
    }
}

int main() {
    vector<int> arr = {3, 1, 4, 1, 5};
    bubbleSort(arr);
    for (int num : arr) cout << num << " "; // 输出1 1 3 4 5
    return 0;
}

```

（6）指数复杂度 $O(2^n)$、阶乘复杂度 O(n!)：效率极差

核心特点：执行时间随n增长呈爆炸式增长，n=20时，2^{20}已达100多万；n=30时，2^{30}达10亿，几乎无法用于实际场景，仅见于极端简单的递归（如斐波那契递归）。

三、空间复杂度：算法“占多少内存”的衡量标准

3.1 核心定义

空间复杂度（Space Complexity），指的是**算法执行所需的额外空间，随数据量n变化的趋势**，同样用大O符号表示。这里的“额外空间”，不包括输入数据本身占用的空间，仅指算法执行过程中临时开辟的空间（如临时变量、数组、栈帧等）。

核心目的：判断算法的内存消耗，尤其是在内存紧张的场景（如嵌入式开发），空间复杂度越低，算法越节省资源。

3.2 怎么判断空间复杂度?

空间复杂度的判断比时间复杂度更简单，重点关注“算法临时开辟的空间规模”，同样遵循“忽略常数、低次项、系数”的原则，核心看3点：

1. 临时变量：若临时变量数量固定，与n无关，空间复杂度为O(1)；
2. 临时数组/容器：若开辟的数组、容器大小与n成正比，空间复杂度为O(n)；
3. 递归栈：递归算法会开辟栈帧存储递归信息，递归深度与n成正比时，空间复杂度为O(n)（如斐波那契递归）。

3.3 常见空间复杂度及案例

（1）常数空间 O(1)：最节省内存

核心特点：算法临时开辟的空间固定，与数据量n无关，不管n多大，占用的额外内存都一样。

案例：两数交换、累加求和、原地排序（如冒泡排序，仅用临时变量存储交换值）。

```
#include <iostream>
using namespace std;

// 两数交换：仅用1个临时变量，空间复杂度O(1)
void swap(int& a, int& b) {
    int temp = a; // 临时变量，数量固定
    a = b;
    b = temp;
}

int main() {
    int a = 3, b = 5;
    swap(a, b);
    cout << a << " " << b << endl; // 输出5 3
    return 0;
}

```

（2）线性空间 O(n)：常用空间复杂度

核心特点：临时开辟的空间与n成正比，n翻倍，额外内存也翻倍，常见于开辟临时数组、容器，或递归深度为n的递归算法。

案例：非原地排序（如归并排序，需开辟临时数组存储拆分后的元素）、用数组存储中间结果。

```cpp
#include <iostream>
#include <vector>
using namespace std;

// 用临时数组存储结果，空间复杂度O(n)
vector<int> doubleArray(vector<int>& arr) {
    int n = arr.size();
    vector<int> temp(n); // 临时数组，大小为n，与数据量成正比
    for (int i = 0; i < n; i++) {
        temp[i] = arr[i] * 2;
    }
    return temp;
}

int main() {
    vector<int> arr = {1, 2, 3};
    vector<int> res = doubleArray(arr);
    for (int num : res) cout << num << " "; // 输出2 4 6
    return 0;
}

```

（3）平方空间 $O(n^2)$：较少使用

核心特点：临时开辟的空间与n^2成正比，内存消耗较大，仅适用于小数据量场景，常见于开辟二维临时数组（如动态规划的二维dp数组）。

案例：用二维数组存储矩阵、动态规划求解最长公共子序列（未优化版本）。

四、关键补充：时间与空间的“取舍”

多数情况下，算法的时间复杂度和空间复杂度是“相互矛盾”的，即“时间换空间”或“空间换时间”，需根据场景选型：

1. 空间换时间：牺牲额外内存，提升运行速度。比如，用哈希表存储数据，查询时间从O(n)降至O(1)，但额外占用O(n)的空间；
2. 时间换空间：牺牲运行速度，节省内存。比如，原地排序（冒泡、选择）比非原地排序（归并）节省O(n)的空间，但时间复杂度更高（$O(n^2)$ vs O(nlogn)）。

实际开发中，优先选“时间换空间”（除非内存极度紧张）——因为用户更在意程序运行速度，而内存资源相对充足；嵌入式、物联网等内存受限场景，需优先优化空间复杂度。

五、常见认知误区，避开这些坑！

误区1：时间复杂度越低，算法一定越好

纠正：需结合数据量判断。比如，$O(n^2)$的冒泡排序，在n=100的小数据量下，可能比O(nlogn)的快速排序更快（因为快排有递归、分区的额外开销），只有大数据量时，低复杂度的优势才会凸显。

误区2：空间复杂度包含输入数据的空间

纠正：不包含。空间复杂度仅计算“算法临时开辟的额外空间”，输入数据（如传入的数组）是用户提供的，不算算法的额外消耗。

误区3：代码越短，时间/空间复杂度越低

纠正：复杂度与代码长度无关。比如，一行递归代码（斐波那契）的时间复杂度是$O(2^n)$，而多行循环代码（迭代求斐波那契）的时间复杂度是O(n)，代码长短不代表效率高低。

六、总结：一句话搞定两个复杂度

时间复杂度：看“核心操作的执行次数随n的增长趋势”，用大O表示，越平缓效率越高；

空间复杂度：看“临时开辟的额外空间随n的增长趋势”，用大O表示，占用越少越节省内存。

对于初学者来说，不用一开始就纠结复杂算法的复杂度推导，先掌握常见复杂度的案例和判断方法，再结合刷题练习，慢慢就能熟练运用——毕竟，能看懂、会判断、能选型，才是学习复杂度的核心目的，而非死记硬背推导公式。