

复杂度

@M了个J

<https://github.com/CoderMJLee>

<http://cnblogs.com/mjios>

码拉松



实力IT教育 www.520it.com

什么是算法

- 算法是用于解决特定问题的一系列的执行步骤

```
// 计算a跟b的和
public static int plus(int a, int b) {
    return a + b;
}

// 计算1+2+3+...+n的和
public static int sum(int n) {
    int result = 0;
    for (int i = 1; i <= n; i++) {
        result += i;
    }
    return result;
}
```

- 使用不同算法，解决同一个问题，效率可能相差非常大
- 比如：求第n个斐波那契数 (fibonacci number)

如何评判一个算法的好坏？

```
// 计算1+2+3+...+n的和
public static int sum1(int n) {
    int result = 0;
    for (int i = 1; i <= n; i++) {
        result += i;
    }
    return result;
}

// 计算1+2+3+...+n的和
public static int sum2(int n) {
    return (1 + n) * n / 2;
}
```

- 如果单从执行效率上进行评估，可能会想到这么一种方案
 - 比较不同算法对同一组输入的执行处理时间
 - 这种方案也叫做：事后统计法
- 上述方案有比较明显的缺点
 - 执行时间严重依赖硬件以及运行时各种不确定的环境因素
 - 必须编写相应的测算代码
 - 测试数据的选择比较难保证公正性
- 一般从以下维度来评估算法的优劣
 - 正确性、可读性、健壮性（对不合理输入的反应能力和处理能力）
 - **时间复杂度**（time complexity）：估算程序指令的执行次数（执行时间）
 - **空间复杂度**（space complexity）：估算所需占用的存储空间

大O表示法 (Big O)

- 一般用大O表示法来描述复杂度，它表示的是数据规模n对应的复杂度

- 忽略常数、系数、低阶

- $9 \gg O(1)$

- $2n + 3 \gg O(n)$

- $n^2 + 2n + 6 \gg O(n^2)$

- $4n^3 + 3n^2 + 22n + 100 \gg O(n^3)$

- 写法上, n^3 等价于 n^3

- 注意：大O表示法仅仅是一种粗略的分析模型，是一种估算，能帮助我们短时间内了解一个算法的执行效率

对数阶的细节

- 对数阶一般省略底数

$$\log_2 n = \log_2 9 * \log_9 n$$

- 所以 $\log_2 n$ 、 $\log_9 n$ 统称为 $\log n$

常见的复杂度

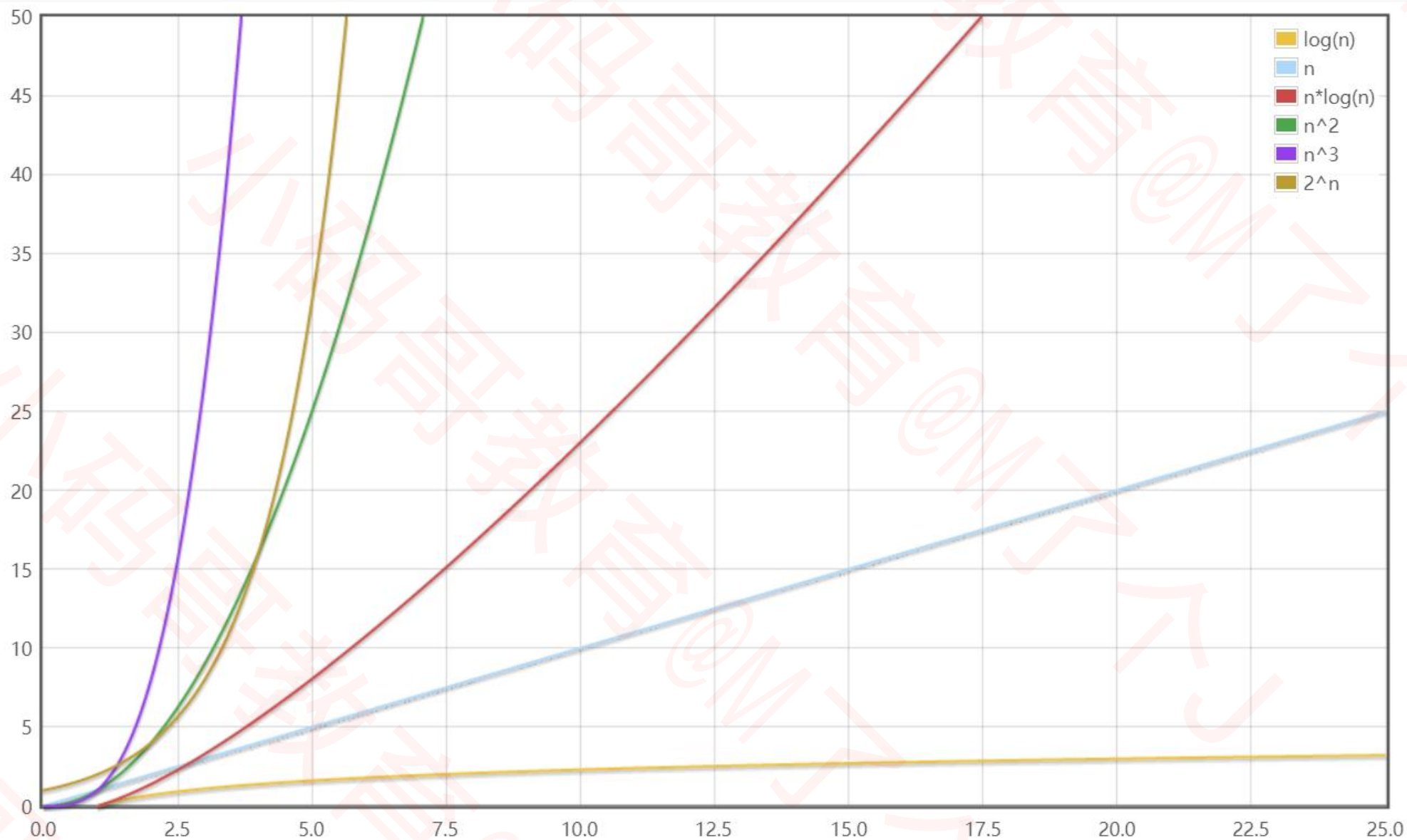
执行次数	复杂度	非正式术语
12	$O(1)$	常数阶
$2n + 3$	$O(n)$	线性阶
$4n^2 + 2n + 6$	$O(n^2)$	平方阶
$4\log_2 n + 25$	$O(\log n)$	对数阶
$3n + 2n\log_3 n + 15$	$O(n\log n)$	$n\log n$ 阶
$4n^3 + 3n^2 + 22n + 100$	$O(n^3)$	立方阶
2^n	$O(2^n)$	指数阶

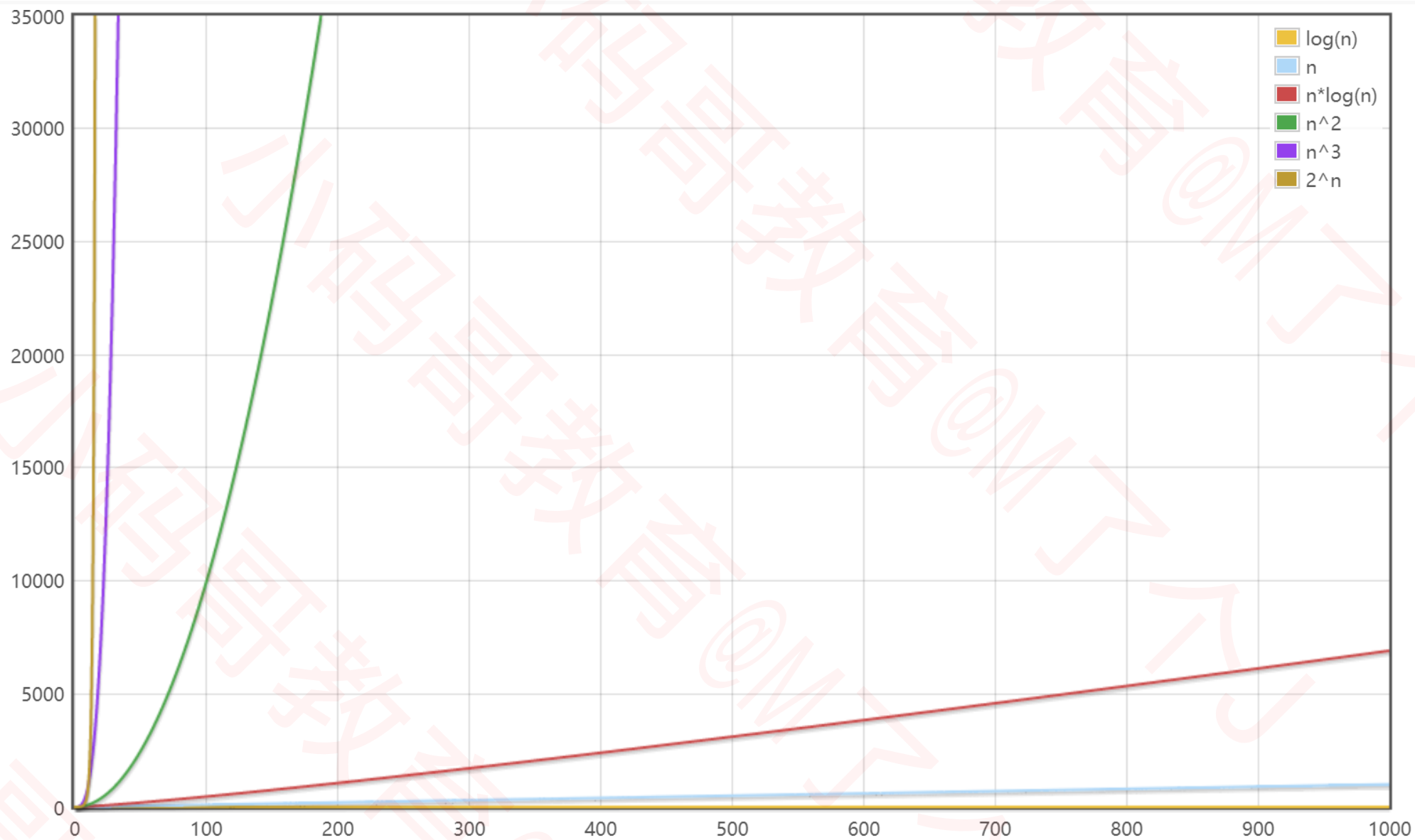
■ $O(1) < O(\log n) < O(n) < O(n\log n) < O(n^2) < O(n^3) < O(2^n) < O(n!) < O(n^n)$

■ 可以借助函数生成工具对比复杂度的大小

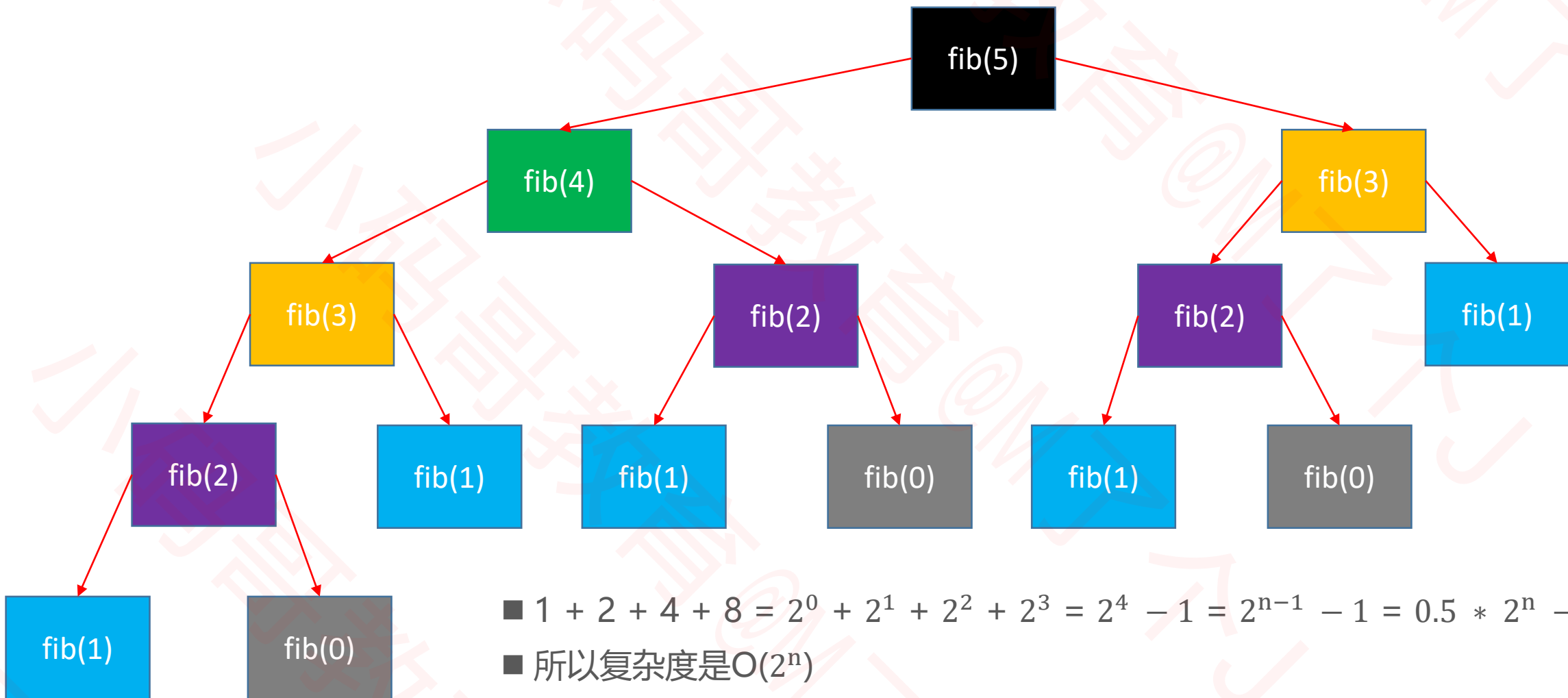
□ <https://zh.numberempire.com/graphingcalculator.php>

数据规模较小时

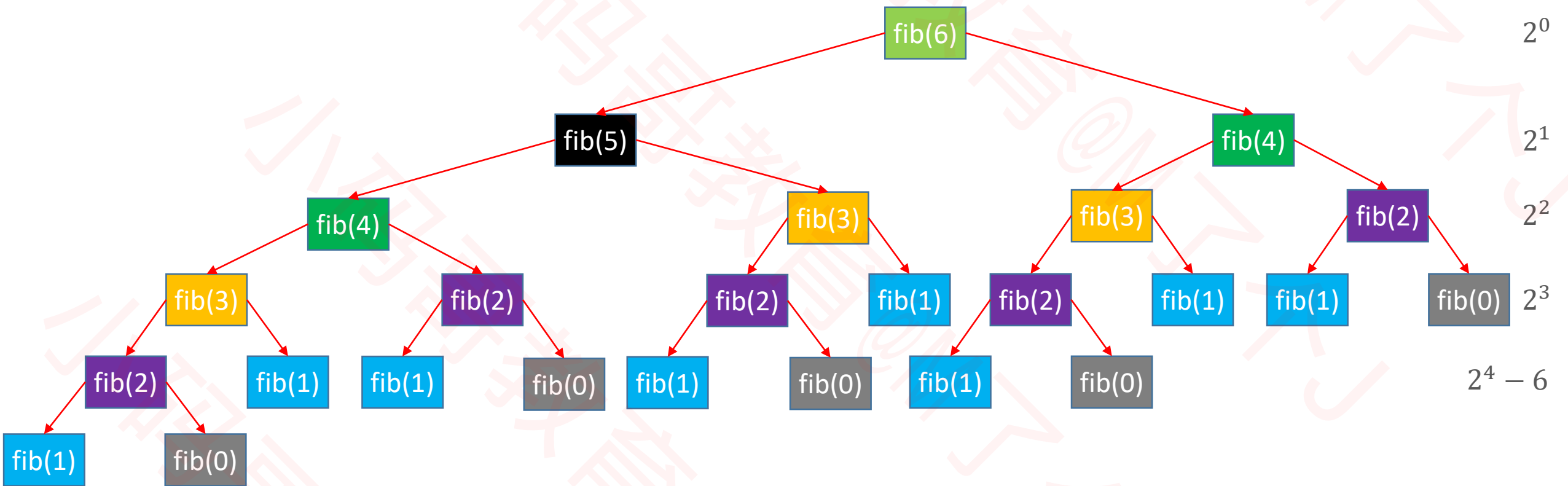




fib函数的时间复杂度分析



fib函数的时间复杂度分析



■ 呈现的是指数级增长的趋势

fib函数的时间复杂度分析

$O(2^n)$

```
public static int fib1(int n) {  
    if (n <= 1) return n;  
    return fib1(n - 2) + fib1(n - 1);  
}
```

$O(n)$

```
public static int fib2(int n) {  
    if (n <= 1) return n;  
  
    int first = 0;  
    int second = 1;  
    while (n-- > 1) {  
        second += first;  
        first = second - first;  
    }  
    return second;  
}
```

- 他们的差别有多大?
- 如果有一台1GHz的普通计算机，运算速度 10^9 次每秒（n为64）
- $O(n)$ 大约耗时 $6.4 * 10^{-8}$ 秒
- $O(2^n)$ 大约耗时584.94年
- 有时候算法之间的差距，往往比硬件方面的差距还要大

- Something interesting
- 我是一个斐波那契程序员
- 因为我每天都在改昨天和前天的bug

算法的优化方向

- 用尽量少的存储空间
- 用尽量少的执行步骤（执行时间）
- 根据情况，可以
 - 空间换时间
 - 时间换空间

多个数据规模的情况

```
public static void test(int n, int k) {  
    for (int i = 0; i < n; i++) {  
        System.out.println("test");  
    }  
  
    for (int i = 0; i < k; i++) {  
        System.out.println("test");  
    }  
}
```

$O(n + k)$

更多知识

- 更多复杂度相关的知识，会在后续讲解数据结构、算法的过程中穿插
- 最好、最坏复杂度
- 均摊复杂度
- 复杂度震荡
- 平均复杂度
-

- 一个用于练习算法的好网站

- <https://leetcode.com/>

- <https://leetcode-cn.com/>

- 斐波那契数

- <https://leetcode-cn.com/problems/fibonacci-number/>

斐波那契的线性代数解法 – 特征方程

$$F(n) = c_1 x_1^n + c_2 x_2^n, \quad x_1 = \frac{1 + \sqrt{5}}{2}, x_2 = \frac{1 - \sqrt{5}}{2}, \quad c_1 = \frac{1}{\sqrt{5}}, c_2 = -\frac{1}{\sqrt{5}}.$$

$$F(n) = \frac{1}{\sqrt{5}} \left[\left(\frac{1 + \sqrt{5}}{2} \right)^n - \left(\frac{1 - \sqrt{5}}{2} \right)^n \right].$$

```
public static int fib3(int n) {  
    double c = Math.sqrt(5);  
    return (int)((Math.pow((1 + c) / 2, n) - Math.pow((1 - c) / 2, n)) / c);  
}
```

时间复杂度：视为O(1)