递归（Recursion）

@M了个J

https://github.com/CoderMJLee

http://cnblogs.com/mjios

码拉松

递归（Recursion）

- 递归：函数（方法）直接或间接调用自身。是一种常用的编程技巧

```
int sum(int n) {
    if (n <= 1) return n;
    return n + sum(n - 1);
}
```

```
void a(int v) {
    if (v < 0) return;
    b(--v);
}

void b(int v) {
    a(--v);
}
```

递归现象

从前有座山，山里有座庙，庙里有个老和尚，正在给小和尚讲故事呢！故事是什么呢？【从前有座山，山里有座庙，庙里有个老和尚，正在给小和尚讲故事呢！故事是什么呢？『从前有座山，山里有座庙，庙里有个老和尚，正在给小和尚讲故事呢！故事是什么呢？……』】

GNU 是 GNU is Not Unix 的缩写

GNU → GNU is Not Unix → GNU is Not Unix is Not Unix → GNU is Not Unix is Not Unix is Not Unix

假设A在一个电影院，想知道自己坐在哪一排，但是前面人很多，
A 懒得数，于是问前一排的人 B【你坐在哪一排？】，只要把 B 的答案加一，就是 A 的排数。
B 懒得数，于是问前一排的人 C【你坐在哪一排？】，只要把 C 的答案加一，就是 B 的排数。
C 懒得数，于是问前一排的人 D【你坐在哪一排？】，只要把 D 的答案加一，就是 C 的排数。
……
直到问到最前面的一排，最后大家都知道自己在哪一排了

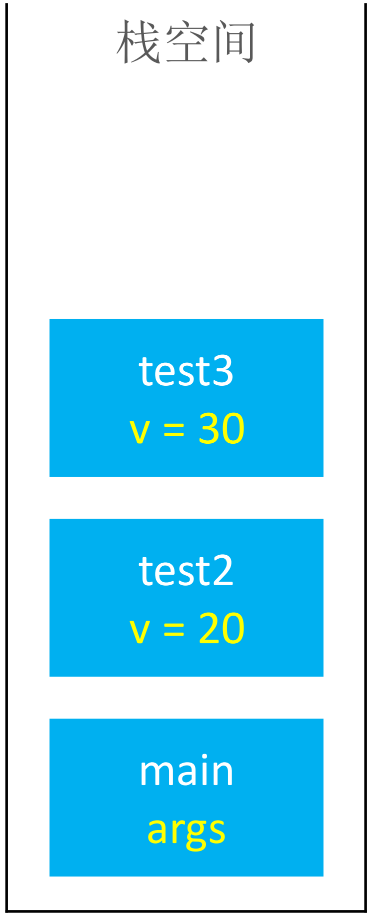
函数的调用过程

```java
public static void main(String[] args) {
    test1(10);
    test2(20);
}

private static void test1(int v) {}

private static void test2(int v) {
    test3(30);
}

private static void test3(int v) {}
```



函数的递归调用过程

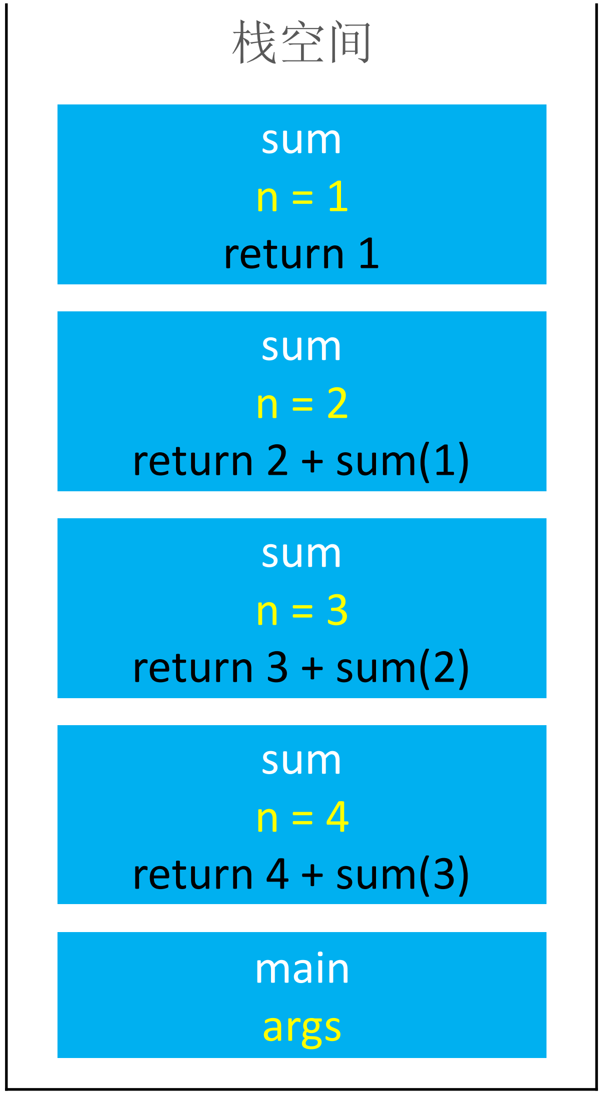
```java
public static void main(String[] args) {
    sum(4);
}

private static int sum(int n) {
    if (n <= 1) return n;
    return n + sum(n - 1);
}
```

- ■ 如果递归调用没有终止，将会一直消耗栈空间
- ☐ 最终导致栈内存溢出（Stack Overflow）

- ■ 所以必需要有一个明确的结束递归的条件
- ☐ 也叫作边界条件、递归基

空间复杂度：$O(n)$



函数的递归调用过程

sum(4)

4 + sum(3)

3 + sum(2)

2 + sum(1)

return 1

4 + 3 + 2 + 1 = 10

3 + 2 + 1

2 + 1

1

实例分析

- 求 1+2+3+...+(n-1)+n 的和（n>0）

```
int sum(int n) {
    if (n <= 1) return n;
    return n + sum(n - 1);
}
```

- 总消耗时间 $T(n) = T(n-1) + O(1)$，因此
- 时间复杂度：$O(n)$
- 空间复杂度：$O(n)$

```
int sum(int n) {
    int result = 0;
    for (int i = 1; i <= n; i++) {
        result += i;
    }
    return result;
}
```

- 时间复杂度：$O(n)$，空间复杂度：$O(1)$

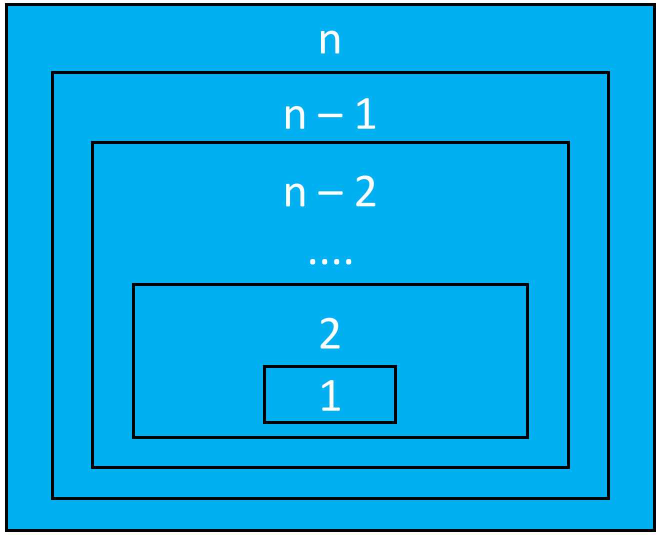
```
int sum(int n) {
    if (n <= 1) return n;
    return (1 + n) * n >> 1;
}
```

- 时间复杂度：$O(1)$，空间复杂度：$O(1)$

- 注意：使用递归不是为了求得最优解，是为了简化解决问题的思路，代码会更加简洁
- 递归求出来的很有可能不是最优解，也有可能是最优解

递归的基本思想

- 拆解问题
  - 把规模大的问题变成规模较小的同类型问题
  - 规模较小的问题又不断变成规模更小的问题
  - 规模小到一定程度可以直接得出它的解



- 求解
  - 由最小规模问题的解得出较大规模问题的解
  - 由较大规模问题的解不断得出规模更大问题的解
  - 最后得出原来问题的解

- 凡是可以利用上述思想解决问题的，都可以尝试使用递归
  - 很多链表、二叉树相关的问题都可以使用递归来解决
  - ✓ 因为链表、二叉树本身就是递归的结构（链表中包含链表，二叉树中包含二叉树）

递归的使用套路

① 明确函数的功能

- 先不要去思考里面代码怎么写，首先搞清楚这个函数的干嘛用的，能完成什么功能？

② 明确原问题与子问题的关系

- 寻找 f(n) 与 f(n – 1) 的关系

③ 明确递归基（边界条件）

- 递归的过程中，子问题的规模在不断减小，当小到一定程度时可以直接得出它的解
- 寻找递归基，相当于是思考：问题规模小到什么程度可以直接得出解？

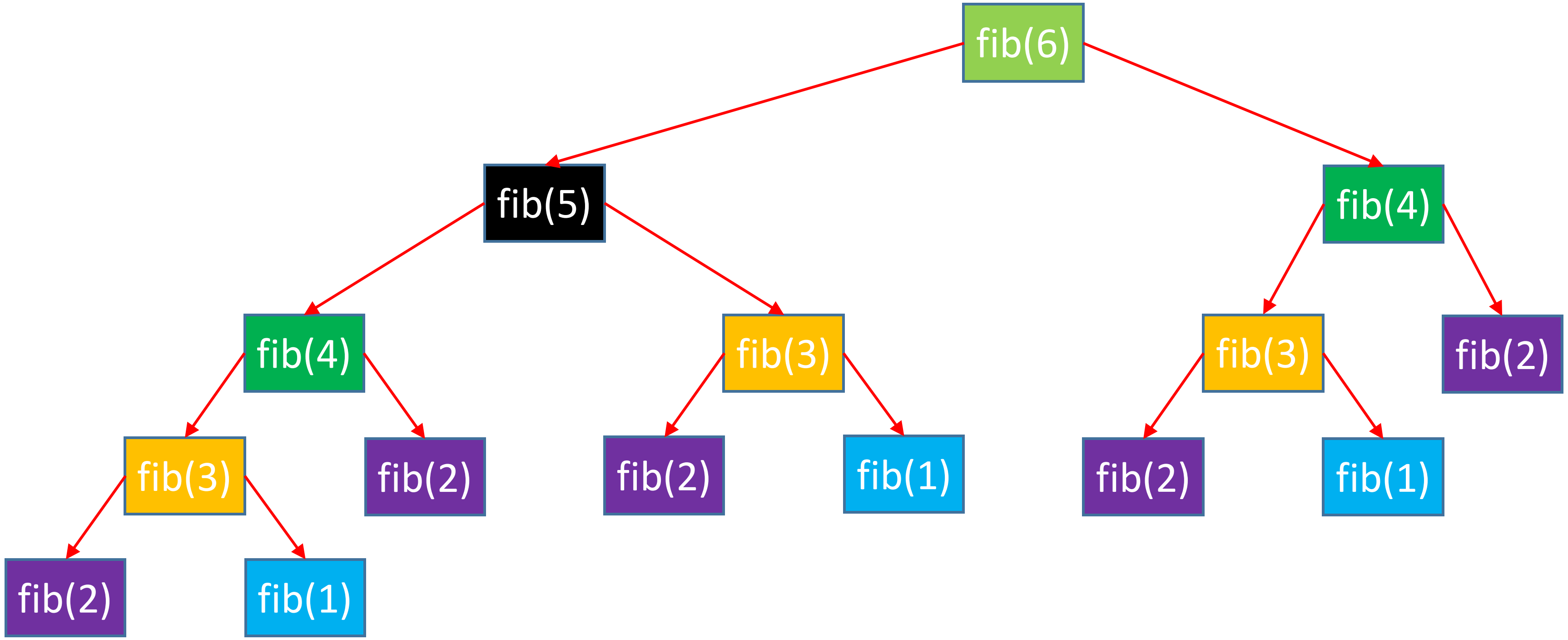
练习1 – 斐波那契数列

- 斐波那契数列：1、1、2、3、5、8、13、21、34、……
  - F(1)=1，F(2)=1, F(n)=F(n-1)+F(n-2)（n≥3）
- 编写一个函数求第 n 项斐波那契数

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

- 根据递推式 $T(n) = T(n-1) + T(n-2) + O(1)$，可得知时间复杂度：$O(2^n)$
- 空间复杂度：$O(n)$
  - 递归调用的空间复杂度 = 递归深度 * 每次调用所需的辅助空间

fib函数的调用过程



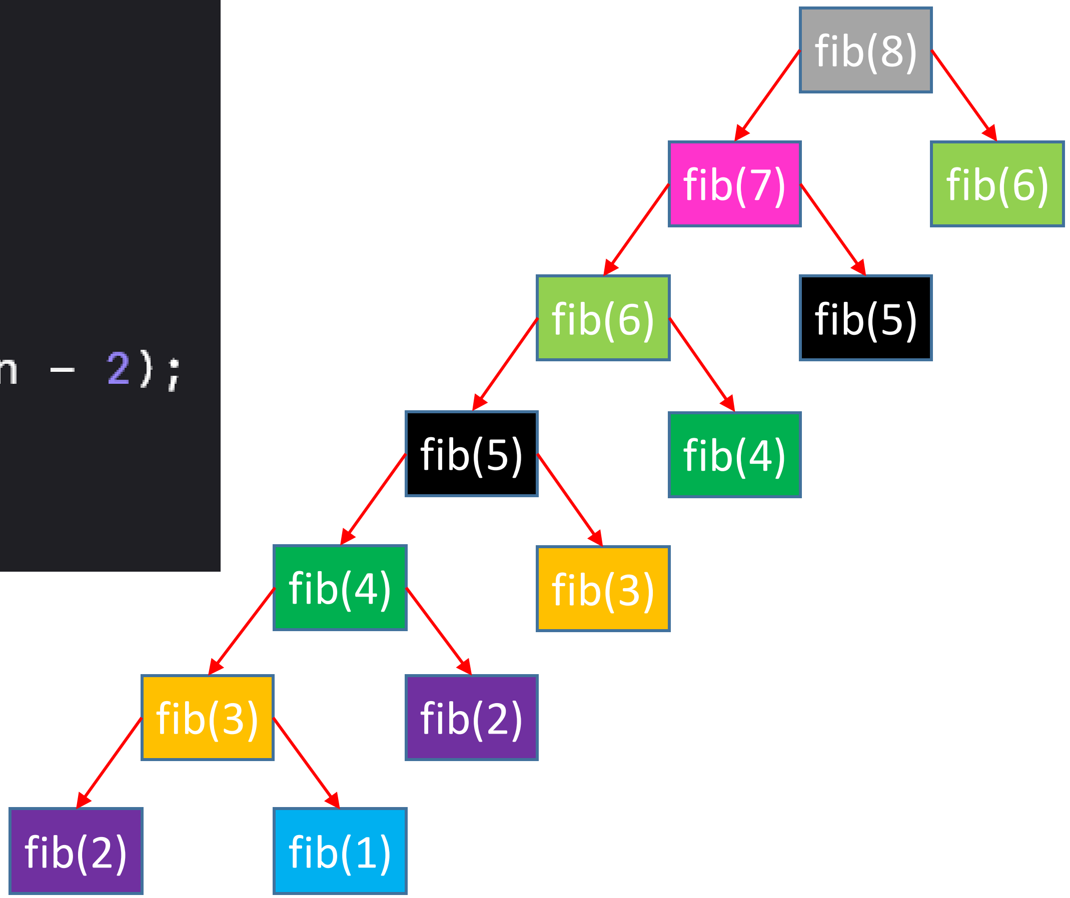
- 出现了特别多的重复计算
- 这是一种“自顶向下”的调用过程

fib优化1 – 记忆化

- 用数组存放计算过的结果，避免重复计算

```
int fib(int n) {
    if(n <= 2) return 1;
    int[] array = new int[n + 1];
    array[2] = array[1] = 1;
    return fib(array, n);
}

int fib(int[] array, int n) {
    if (array[n] == 0) {
        array[n] = fib(array, n - 1) + fib(array, n - 2);
    }
    return array[n];
}
```



- 时间复杂度：O(n)，空间复杂度：O(n)

fib优化2

■ 去除递归调用

```
int fib(int n) {
    if (n <= 2) return 1;
    int[] array = new int[n + 1];
    array[2] = array[1] = 1;
    for (int i = 3; i <= n; i++) {
        array[i] = array[i - 1] + array[i - 2];
    }
    return array[n];
}
```

■ 时间复杂度：O(n)，空间复杂度：O(n)

■ 这是一种“自底向上”的计算过程

fib优化3

- 由于每次运算只需要用到数组中的 2 个元素，所以可以使用滚动数组来优化

```
int fib(int n) {
    if (n <= 2) return 1;
    int[] array = new int[2];
    array[0] = array[1] = 1;
    for (int i = 3; i <= n; i++) {
        array[i % 2] = array[(i - 1) % 2] + array[(i - 2) % 2];
    }
    return array[n % 2];
}
```

- 时间复杂度：O(n)，空间复杂度：O(1)

fib优化4 – 位运算取代模运算

■ 乘、除、模运算效率较低，建议用其他方式取代

```java
int fib(int n) {
    if (n <= 2) return 1;
    int[] array = new int[2];
    array[0] = array[1] = 1;
    for (int i = 3; i <= n; i++) {
        array[i & 1] = array[(i - 1) & 1] + array[(i - 2) & 1];
    }
    return array[n & 1];
}
```

fib优化5

```
int fib(int n) {
    if (n <= 2) return 1;
    int first = 1;
    int second = 1;
    for (int i = 3; i <= n; i++) {
        second = first + second;
        first = second - first;
    }
    return second;
}
```

- 时间复杂度：O(n)，空间复杂度：O(1)

fib优化6

- 斐波那契数列有个线性代数解法：特征方程

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

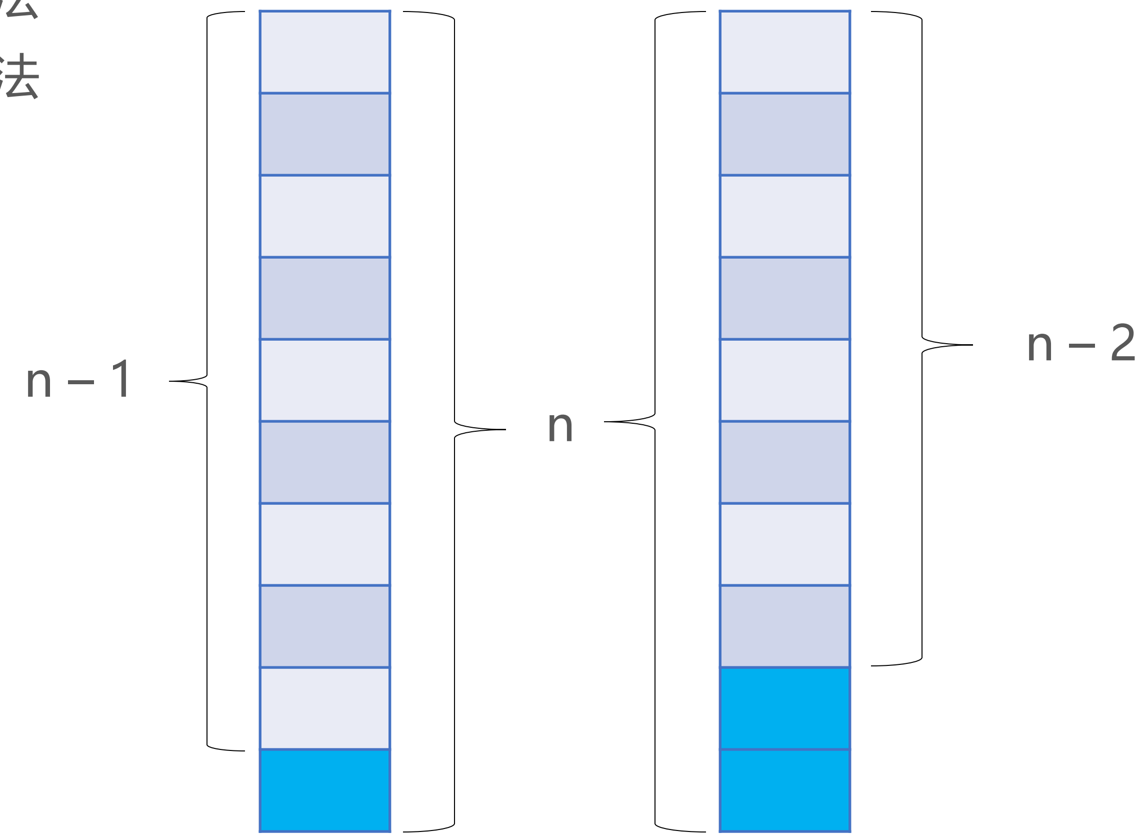
$$F(n) = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right].$$

```
int fib(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、空间复杂度取决于 pow 函数（至少可以低至O(logn)）

练习2 – 上楼梯（跳台阶）

- 楼梯有 n 阶台阶，上楼可以一步上 1 阶，也可以一步上 2 阶，走完 n 阶台阶共有多少种不同的走法？
- 假设 n 阶台阶有 f(n) 种走法，第 1 步有 2 种走法
  - ✓ 如果上 1 阶，那就还剩 n – 1 阶，共 f(n – 1) 种走法
  - ✓ 如果上 2 阶，那就还剩 n – 2 阶，共 f(n – 2) 种走法
- 所以 f(n) = f(n – 1) + f(n – 2)



练习2 – 上楼梯

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

- 跟斐波那契数列几乎一样，因此优化思路也是一致的

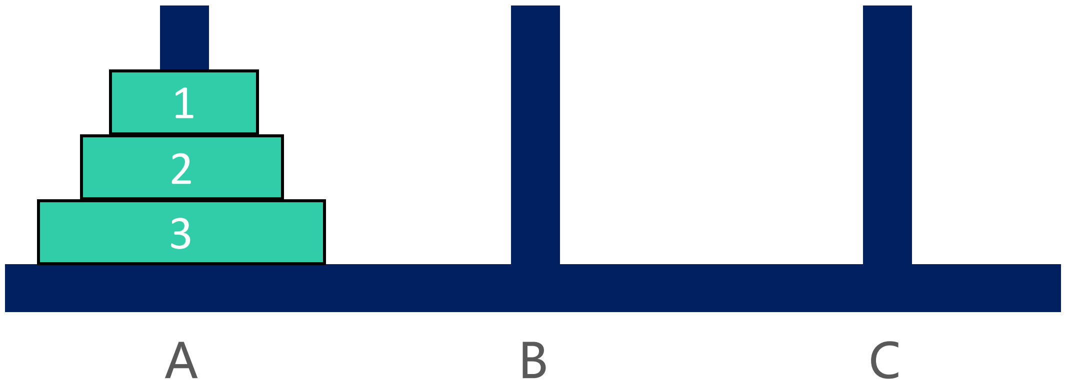
```
int climbStairs(int n) {
    if (n <= 2) return n;
    int first = 1;
    int second = 2;
    for (int i = 3; i <= n; i++) {
        second = first + second;
        first = second - first;
    }
    return second;
}
```

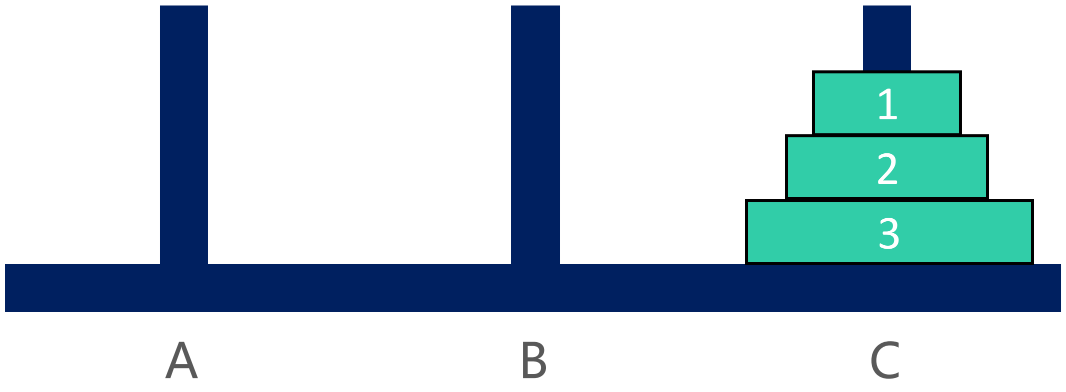
练习3 – 汉诺塔（Hanoi）

■ 编程实现把 A 的 n 个盘子移动到 C（盘子编号是 [1, n] ）

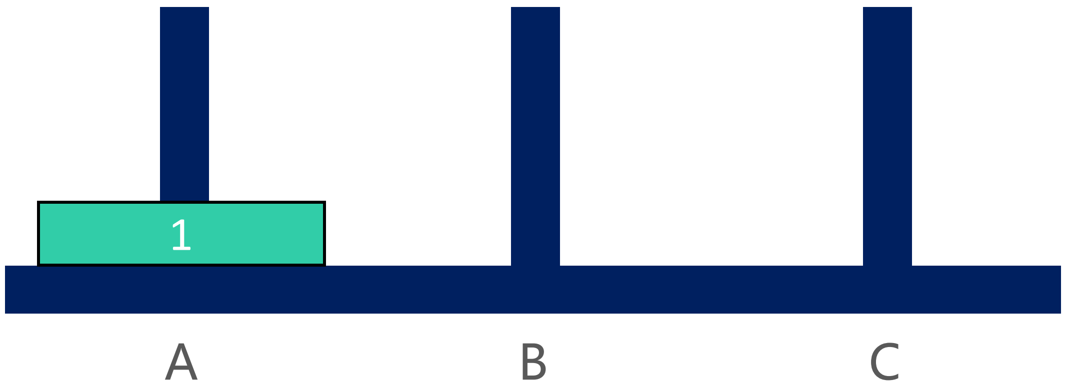
□ 每次只能移动1个盘子

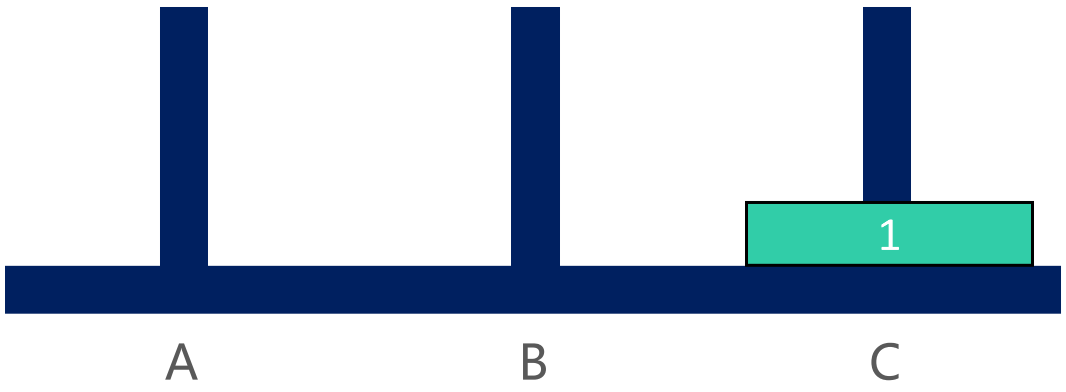
□ 大盘子只能放在小盘子下面



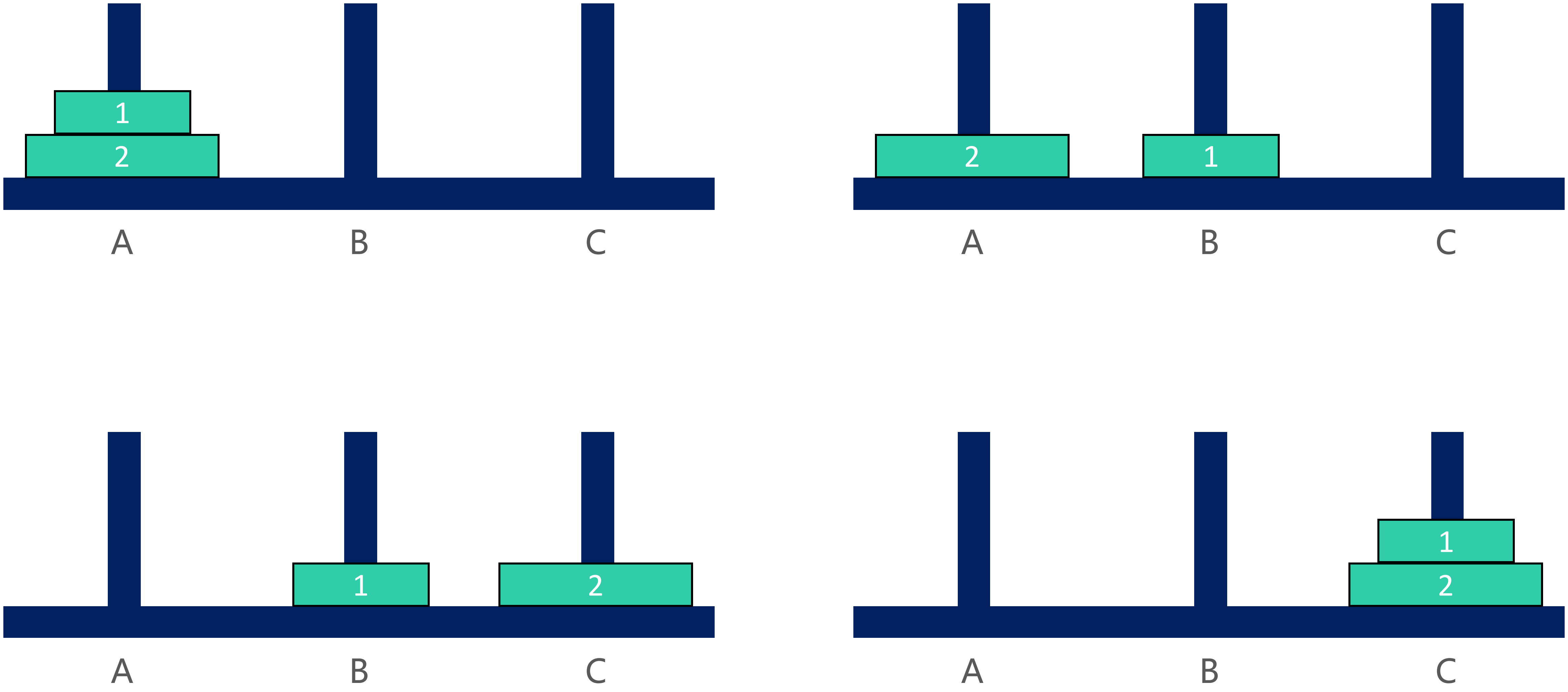


1个盘子

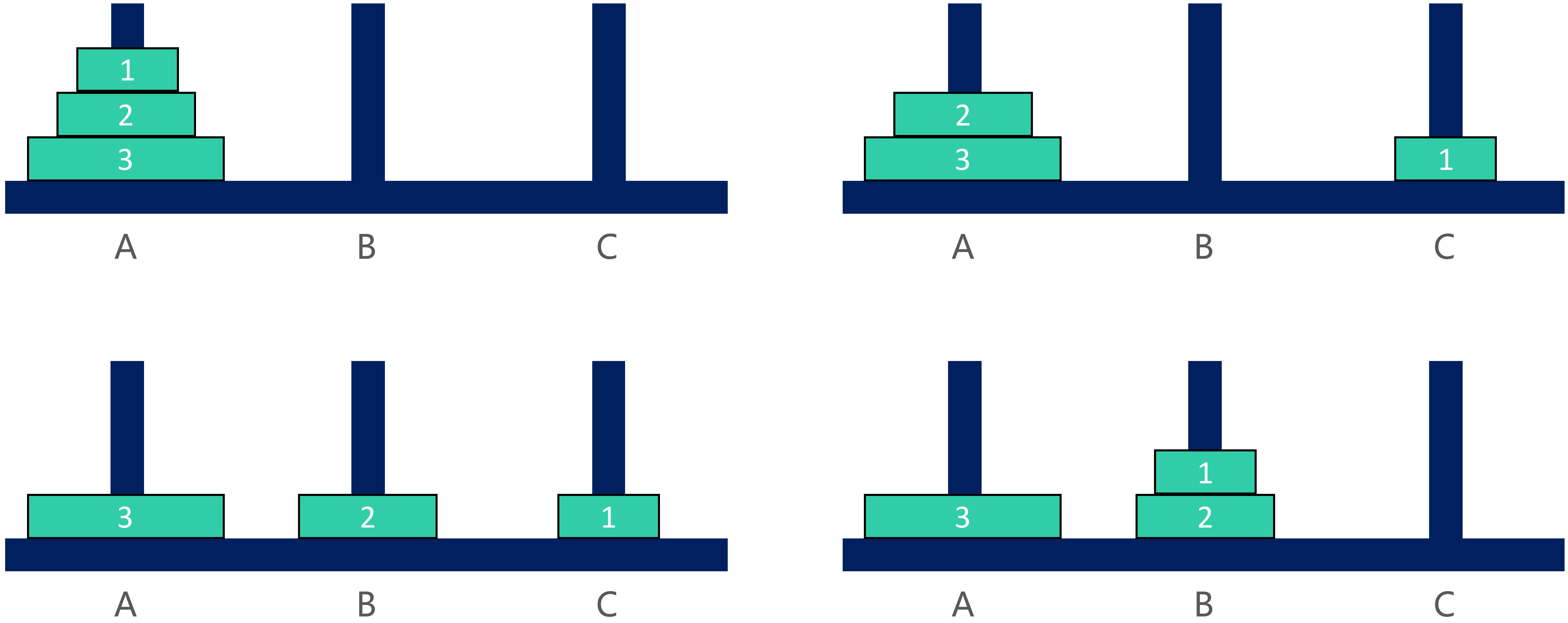




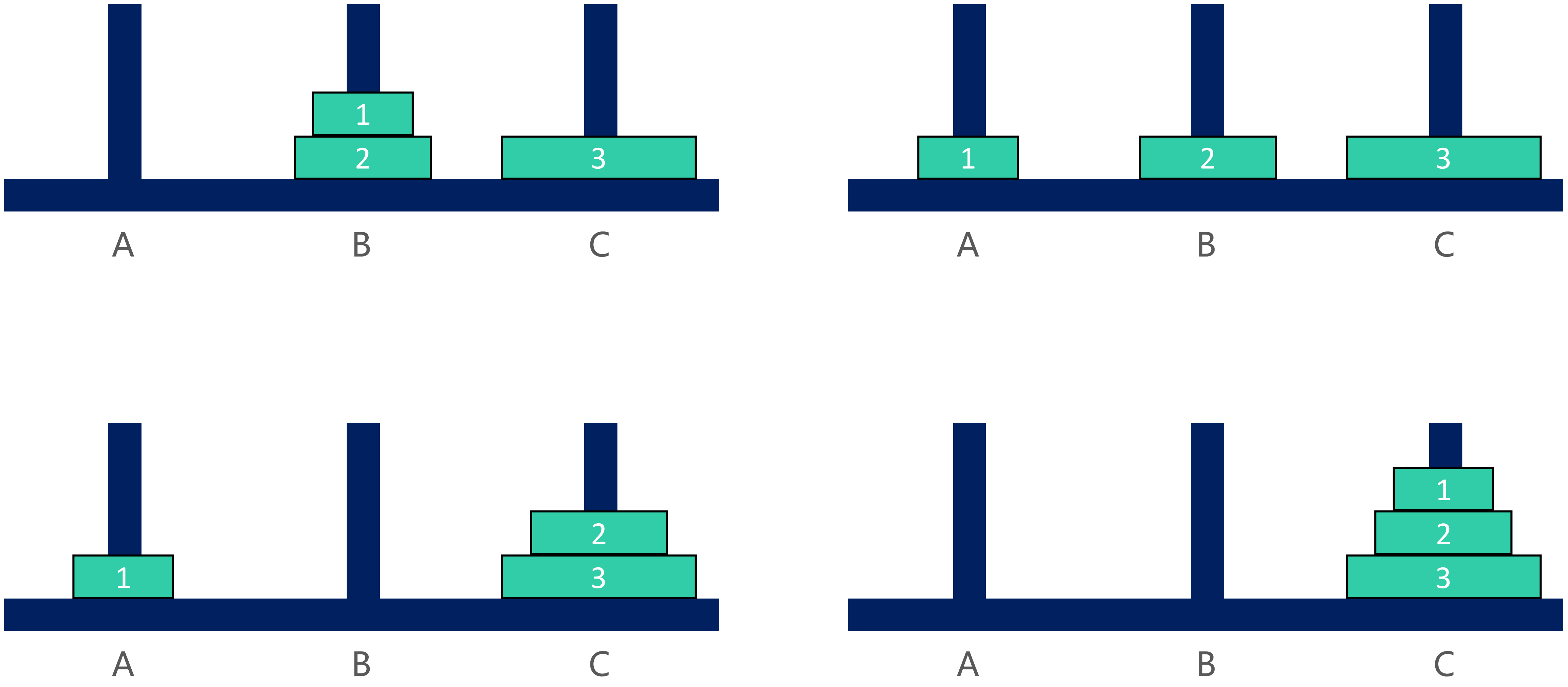
2个盘子



3个盘子



3个盘子



汉诺塔 – 思路

■ 其实分 2 种情况讨论即可

□ 当 n == 1时，直接将盘子从 A 移动到 C

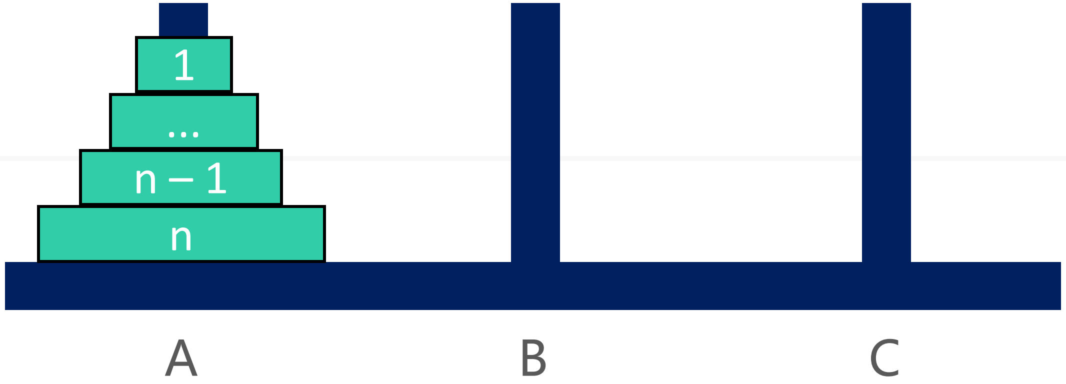
□ 当 n > 1时，可以拆分成3大步骤

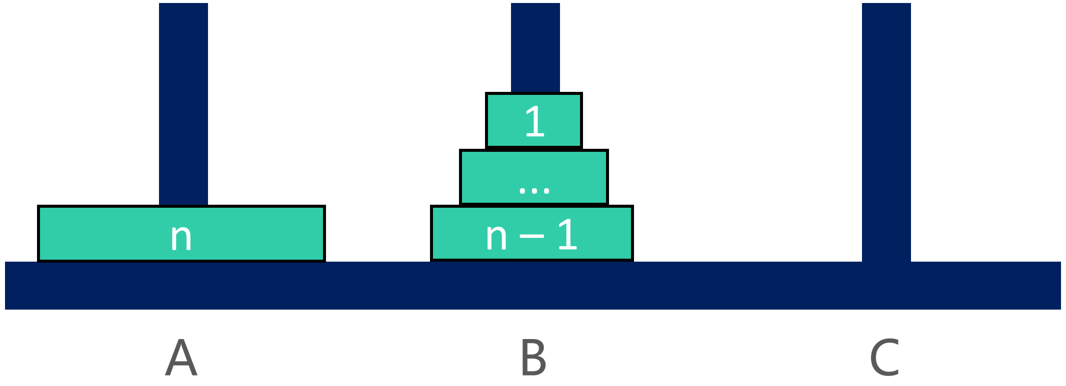
① 将 n – 1 个盘子从 A 移动到 B

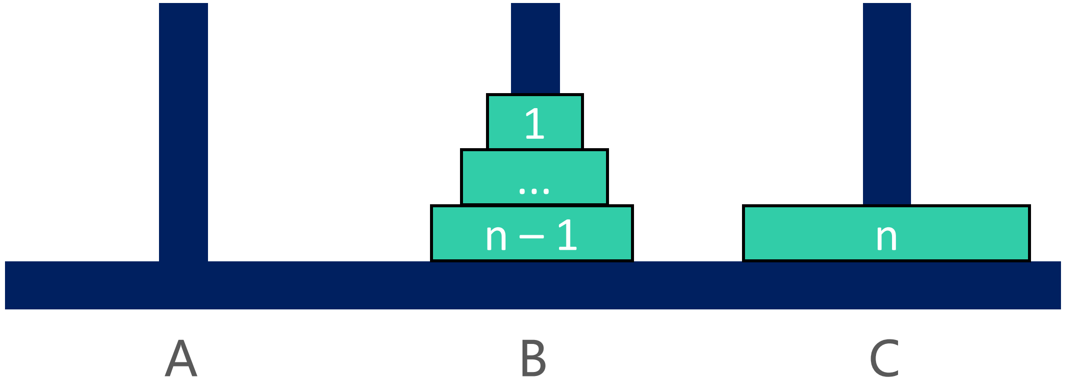
② 将编号为 n 的盘子从 A 移动到 C

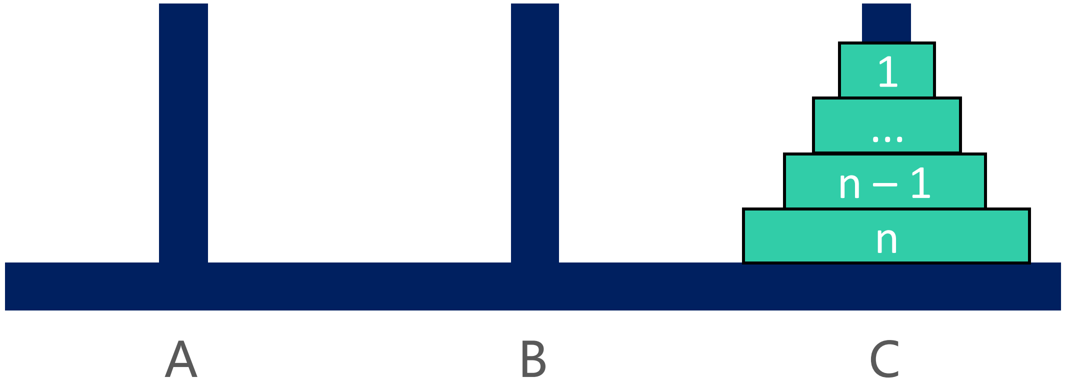
③ 将 n – 1 个盘子从 B 移动到 C

✓ 步骤 ① ③ 明显是个递归调用









汉诺塔 – 实现

```java
/**
 * 将第 i 号盘子从 from 移动到 to
 */
void move(int i, String from, String to) {
    System.out.println(i + "号盘子: " + from + "->" + to);
}
```

```java
/**
 * 将 n 个盘子从 p1 移动到 p3
 */
void hanoi(int n, String p1, String p2, String p3) {
    if (n <= 1) {
        move(n, p1, p3);
        return;
    }
    hanoi(n - 1, p1, p3, p2);
    move(n, p1, p3);
    hanoi(n - 1, p2, p1, p3);
}
```

- $T(n) = 2 * T(n-1) + O(1)$
- 因此时间复杂度是：$O(2^n)$

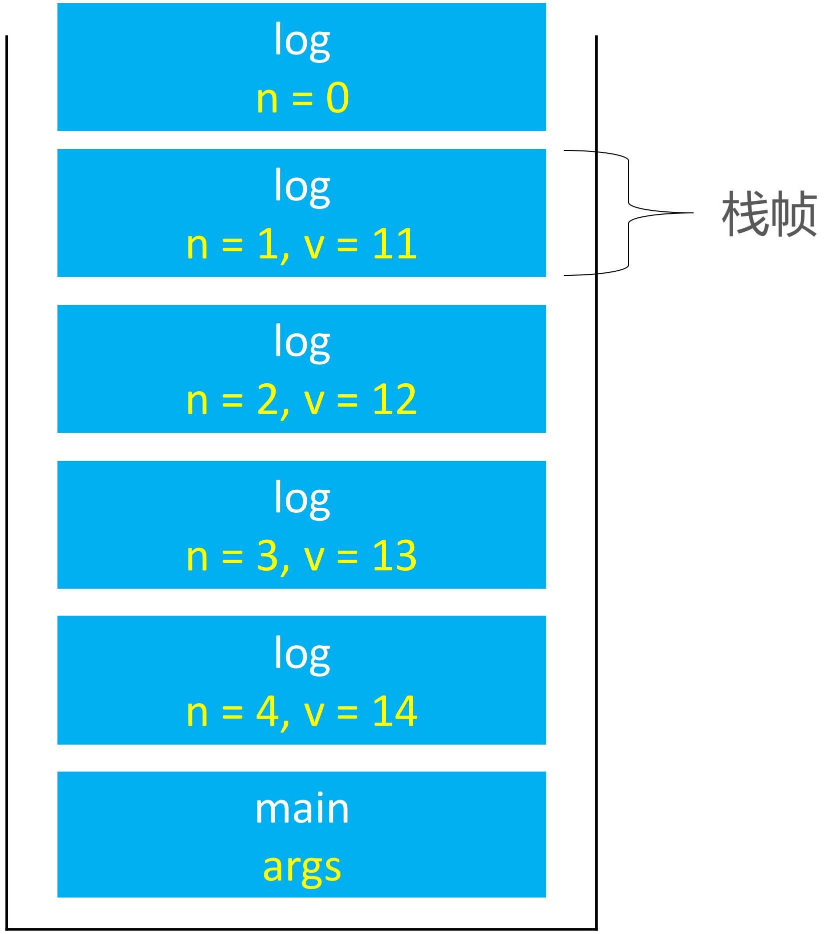
- 空间复杂度：$O(n)$

递归转非递归

- 递归调用的过程中，会将每一次调用的参数、局部变量都保存在了对应的栈帧（Stack Frame）中

```java
public static void main(String[] args) {
    log(4);
}

static void log(int n) {
    if (n < 1) return;
    log(n - 1);
    int v = n + 10;
    System.out.println(v);
}
```



- 若递归调用深度较大，会占用比较多的栈空间，甚至会导致栈溢出
- 在有些时候，递归会存在大量的重复计算，性能非常差
- 这时可以考虑将递归转为非递归（递归100%可以转换成非递归）

递归转非递归

- 递归转非递归的万能方法
  - 自己维护一个栈，来保存参数、局部变量
  - 但是空间复杂度依然没有得到优化

```java
static class Frame {
    int n;
    int v;
    Frame(int n, int v) {
        this.n = n;
        this.v = v;
    }
}
```

```java
static void log(int n) {
    Stack<Frame> frames = new Stack<>();
    while (n > 0) {
        frames.push(new Frame(n, n + 10));
        n--;
    }
    while (!frames.isEmpty()) {
        Frame frame = frames.pop();
        System.out.println(frame.v);
    }
}
```

递归转非递归

- 在某些时候，也可以重复使用一组相同的变量来保存每个栈帧的内容

```java
static void log(int n) {
    for (int i = 1; i <= n; i++) {
        System.out.println(i + 10);
    }
}
```

- 这里重复使用变量 i 保存原来栈帧中的参数
- 空间复杂度从 O(n) 降到了 O(1)

尾调用（Tail Call）

- 尾调用：一个函数的最后一个动作是调用函数
  - 如果最后一个动作是调用自身，称为尾递归（Tail Recursion），是尾调用的特殊情况

```c
void test1() {
    int a = 10;
    int b = a + 20;
    test2(b);
}
```

```c
void test2(int n) {
    if (n < 0) return;
    test2(n - 1);
}
```

- 一些编译器能对尾调用进行优化，以达到节省栈空间的目的

下面代码不是尾调用

```c
int factorial(int n) {
    if (n <= 1) return n;
    return n * factorial(n - 1);
}
```

- 因为它最后1个动作是乘法

尾调用优化（Tail Call Optimization）

- 尾调用优化也叫做尾调用消除（Tail Call Elimination）
  - 如果当前栈帧上的局部变量等内容都不需要用了，当前栈帧经过适当的改变后可以直接当作被尾调用的函数的栈帧使用，然后程序可以 jump 到被尾调用的函数代码
  - 生成栈帧改变代码与 jump 的过程称作尾调用消除或尾调用优化
  - 尾调用优化让位于尾位置的函数调用跟 goto 语句性能一样高

- 消除尾递归里的尾调用比消除一般的尾调用容易很多
  - 比如Java虚拟机（JVM）会消除尾递归里的尾调用，但不会消除一般的尾调用（因为改变不了栈帧）
  - 因此尾递归优化相对比较普遍，平时的递归代码可以考虑尽量使用尾递归的形式

尾调用优化前的汇编代码（C++）

```cpp
void test(int n) {
    if (n < 0) return;
    printf("test - %d\n", n);
    test(n - 1);
}
```

```
void test(int n) {
010C1080  push        ebp
010C1081  mov         ebp,esp
    if (n < 0) return;
010C1083  cmp         dword ptr [n],0
010C1087  jge         test+0Bh (010C108Bh)
010C1089  jmp         test+2Bh (010C10ABh)
    printf("test - %d\n", n);
010C108B  mov         eax,dword ptr [n]
010C108E  push        eax
010C108F  push        10C2104h
010C1094  call        printf (010C1040h)
010C1099  add         esp,8
    test(n - 1);
010C109C  mov         ecx,dword ptr [n]
010C109F  sub         ecx,1
010C10A2  push        ecx
010C10A3  call        test (010C1080h)
010C10A8  add         esp,4
}
```

尾调用优化后的汇编代码（C++）

```
void test(int n) {
    if (n < 0) return;
    printf("test - %d\n", n);
    test(n - 1);
}
```

```
void test(int n) {
00091050  push        ebp
00091051  mov         ebp,esp
00091053  and         esp,0FFFFFFF8h
00091056  push        ecx
00091057  push        esi
00091058  mov         esi,ecx
    if (n < 0) return;
0009105A  test        esi,esi
0009105C  js          test+23h (091073h)
0009105E  xchg        ax,ax
    printf("test - %d\n", n);
00091060  push        esi
00091061  push        offset string "test - %d\n" (0920F8h)
00091066  call        printf (091010h)
0009106B  add         esp,8
    test(n - 1);
0009106E  sub         esi,1
00091071  jns         test+10h (091060h)
}
```

■ 汇编教程

- https://ke.qq.com/course/348781
- https://ke.qq.com/course/336509

尾递归示例1 – 阶乘

■ 求 n 的阶乘 1*2*3*...*(n-1)*n （n>0）

```
int factorial(int n) {
    if (n <= 1) return n;
    return n * factorial(n - 1);
}
```

```
int factorial(int n) {
    return factorial(n, 1);
}

/**
 * @param result 从大到小累乘的结果
 */
int factorial(int n, int result) {
    if (n <= 1) return result;
    return factorial(n - 1, n * result);
}
```

尾递归示例2 – 斐波那契数列

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

```
int fib(int n) {
    return fib(n, 1, 1);
}

public int fib(int n, int first, int second) {
    if(n <= 1) return first;
    return fib(n - 1, second, first + second);
}
```