

# 第7章 用函数实现模块化程序设计

7.1为什么要用函数

7.2怎样定义函数

7.3调用函数

7.4对被调用函数的声明和函数原型

7.5函数的多级嵌套调用

7.6递归函数设计

7.7数组作为函数参数

7.8局部变量和全局变量

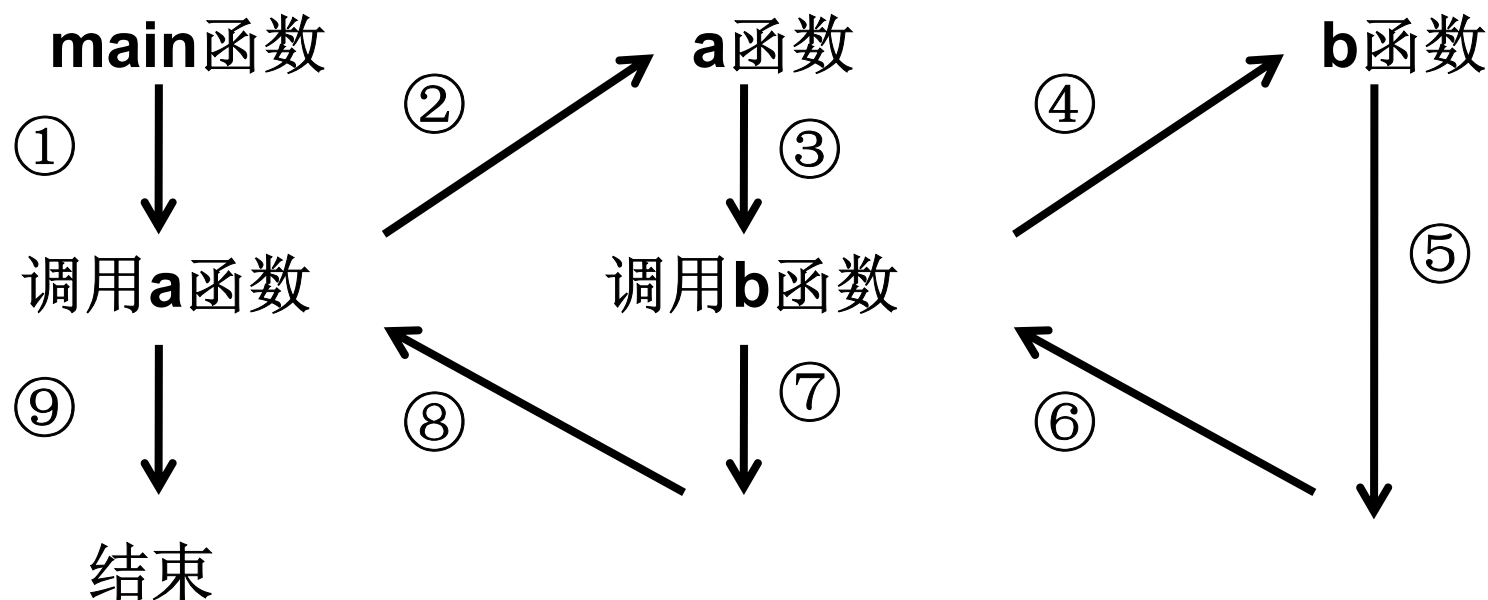
7.9变量的存储类别和生存期

7.10函数说明符

7.11 内部函数和外部函数

## 7.5 函数的多级嵌套调用

- C 语言的函数实现特定的功能，以便其他函数调用。有些功能在一个程序中可能被调用多次，故不建议将一个函数定义在另一个函数内部（函数嵌套）。
- 函数可以多级嵌套调用，即调用一个函数的过程中，又可以调用另一个函数



## 7.5 函数的嵌套调用

例**7.5** 输入**4**个整数，找出其中最大的数。用函数的嵌套调用来处理。

➤ 解题思路：

- ◆ 定义一个求**4**个数最大值的函数，不妨命名为**max4**，然后在**main**中调用**max4**函数，得到**4**数之最大者；
- ◆ 前面已经介绍过求两数最大值的函数**max**，得到两个数中的大者；
- ◆ **max4**中多次调用**max**，即可找到**4**个数中的最大者，然后把它作为函数值返回**main**函数即可；
- ◆ 最后在**main**函数中输出结果

## 主函数

```
#include <stdio.h>
```

```
int main()
```

```
{ int max4(int a,int b,int c,int d);
```

对max4 函数声明

```
int a,b,c,d,maxValue;
```

```
printf("4 interger numbers:");
```

```
scanf("%d%d%d%d",&a,&b,&c,&d);
```

输入4个整数

```
maxValue = max4(a,b,c,d);
```

调用后得到4个数中最大者

```
printf("max=%d \n", maxValue);
```

输出最大者

```
return 0;
```

```
}
```

```
int max4(int a,int b,int c,int d)
```

对max函数声明

```
{ int max(int a,int b);
```

```
int m;
```

m = max(max(a, b), max(c, d));

```
m=max(a,b);
```

```
m=max(m,c);
```

```
m=max(m,d);
```

```
return m;
```

```
}
```

```
int max(int a,int b)
```

```
{
```

```
return a>b?a:b;
```

```
}
```

## 7.6 递归函数设计

- 在调用一个函数的过程中又出现直接或间接地调用该函数本身，称为函数的**递归调用**。C语言的特点之一就在于允许函数的递归调用。
- 递归是计算机科学的一个重要概念，递归算法是程序设计中的有效方法，也是分治法程序设计的基础，能采用递归编写的程序看起来非常简洁和清晰。
- 递归是一种程序设计技巧，也是一种算法设计的策略。
- 递归按照调用方式，可以分为**直接递归**和**间接递归**
  - ◆ 直接递归是指函数在执行过程中直接调用自身；
  - ◆ 间接递归是指函数在执行过程中调用了其他的函数，再经过这些函数调用了自身。
- 从数学的角度来分析，递归的数学模型就是**递推原理**，在整个过程中，反复实现的都是同一个原理或操作，其本质和**数学归纳法**类似。

## 7.6 递归函数设计 ——常见递推式

➤ 自然数的阶乘

◆  $n! = n * (n-1)!$

➤ 斐波那契数列  $F(n) = F(n-1) + F(n-2)$

◆  $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \dots$

➤ 卢卡斯数列  $L(n) = L(n-1) + L(n-2)$

◆  $2, 1, 3, 4, 7, 11, 18, 29, 47, 76, 123, \dots$

➤ 连续自然数求和  $S(n) = \sum_{i=0}^n i = n + S(n-1)$

◆  $0, 1, 3, 6, 10, 15, 21, 28, 36, 45, \dots$

➤ 一般递推式

◆  $a(n) = f(a(n-1), a(n-2), \dots, a(n-p))$

## 7.6 递归函数设计

### ——递归可以求解的问题

➤ 递归适用于下述问题

- ◆ 解决一个问题可以转化为解决它的子问题，而它的子问题又变成子问题的子问题
- ◆ 所有问题的解决都是采用同一个模型，即：
  - 需要解决的问题和它的子问题具有相同的逻辑归纳处理项。
- ◆ 并且有一个子问题是例外的，也就是递归结束的那一项，处理方法不适用于上述的归纳处理项，当然也不能使用这种方法去处理，否则就形成了无穷递归。

➤ 递归的核心

- ◆ 归纳终结点以及直接求解的表达式。

## 7.6 递归函数设计 ——递归求解流程

- 递推求解的流程可以采用如下的列表来表示：
  - ◆ 步进表达式：问题转换为子问题求解的表达式；
  - ◆ 结束条件：不再适用于步进表达式的情况，亦即什么时候不再使用步进表达式(例外)；
  - ◆ 直接求解表达式：在结束条件下能够直接计算返回值的表达式；
  - ◆ 逻辑归纳项：适用于一切非结束条件下子问题的处理，包含上述的步进表达式；



## 7.6 递归函数设计

### ——递归求解必须满足的四个特征

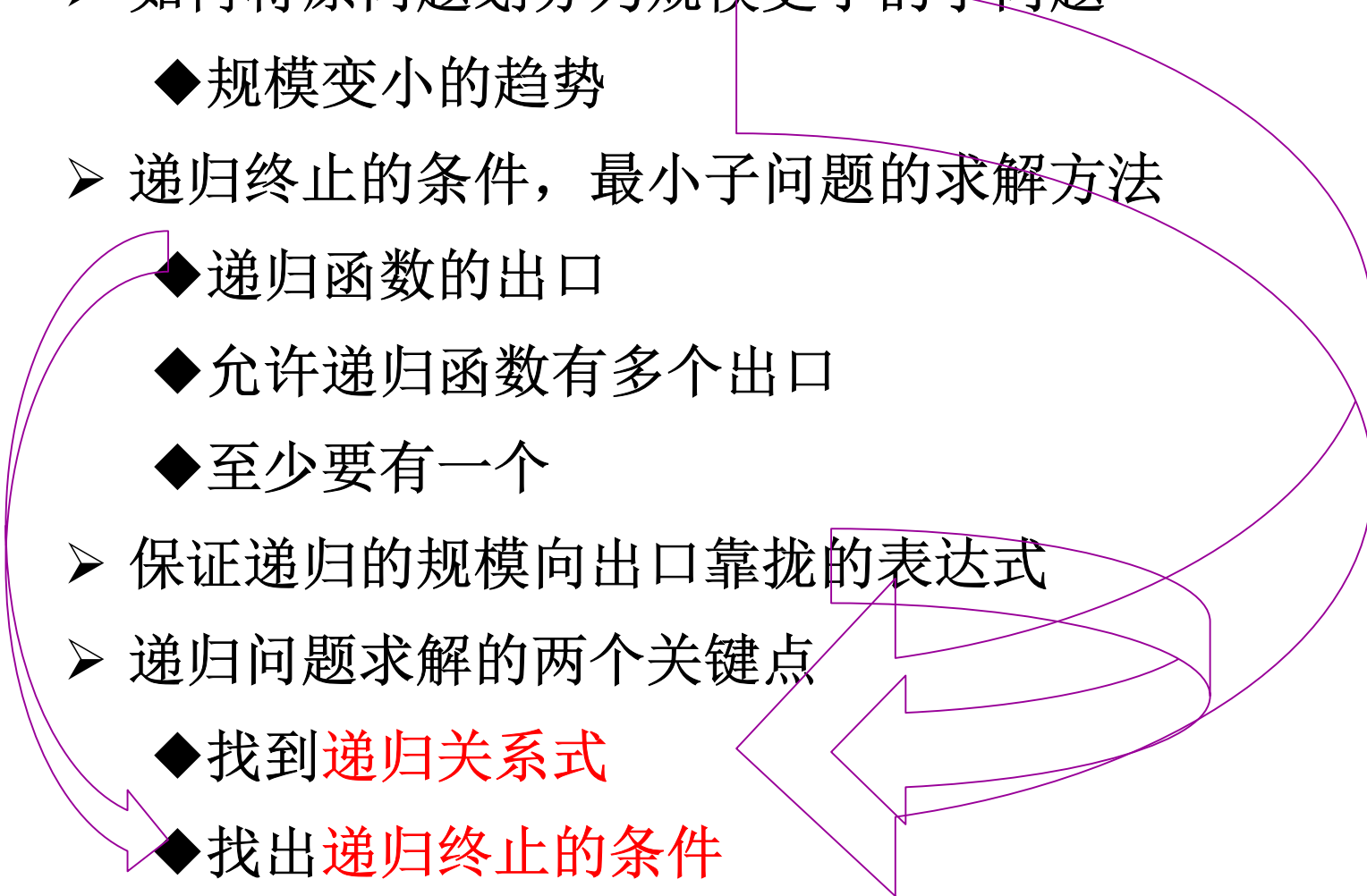
- 必须有可最终达到的终止条件，否则程序将陷入无穷循环
- 子问题的规模要比原问题小，或更接近终止条；
- 子问题可以通过再次递归求解或因满足终止条件而直接求解
- 子问题的解应能组合为整个问题的解。

## 7.6 递归函数设计 ——递归求解方法

- 递归的基本思想是把规模大的问题转化为规模较小的相似的子问题来解决，且这些规模较小的子问题的求解过程相对比较容易，同时规模较小的子问题的解足以构成原问题的解。
- 在函数（算法）实现时，由于解决大问题的方法和解决小问题的方法往往是同一个方法，所以就产生了函数调用其自身的情况。
- 由于出现了自身调用自身的情况，这个解决问题的函数必须有明确的结束条件，也就是说递归函数必须是收敛的，这样才可以避免出现无穷递归的情况。

## 7.6 递归函数设计

### ——设计递归函数必须考虑的问题

- 如何将原问题划分为规模更小的子问题
    - ◆ 规模变小的趋势
  - 递归终止的条件，最小子问题的求解方法
    - ◆ 递归函数的出口
    - ◆ 允许递归函数有多个出口
    - ◆ 至少要有有一个
  - 保证递归的规模向出口靠拢的表达式
  - 递归问题求解的两个关键点
    - ◆ 找到递归关系式
    - ◆ 找出递归终止的条件
- 
- The diagram consists of several purple arrows illustrating the flow of a recursive process. One arrow starts from the '规模变小的趋势' (Trend of decreasing scale) item and points to the '递归终止的条件' (Recursive termination condition) item. Another arrow starts from the '递归终止的条件' item and points to the '递归函数的出口' (Recursive function exit) item. A third arrow starts from the '递归函数的出口' item and points to the '至少要有有一个' (At least have one) item. A fourth arrow starts from the '至少要有有一个' item and points to the '递归关系式' (Recursive relationship) item. A fifth arrow starts from the '递归关系式' item and points to the '递归终止的条件' item, forming a loop. Additionally, there are arrows pointing from the '递归终止的条件' item to the '递归关系式' item and from the '递归关系式' item to the '递归终止的条件' item, suggesting a bidirectional relationship or iterative refinement.

## 7.6 递归函数设计

### ——递归函数的一般形式

➤ 递归函数的一般形式

```
◆ T func(mode){  
  ◆ if(endCondition){           // 满足终止条件的情况  
  ◆   directExpression; // 直接求解表达式  
  ◆ }  
  ◆ else{                       // 非终止条件的情况  
  ◆   inductiveExpression; // 归纳表达式  
  ◆   mode修正;              // 逐步趋于终止条件  
  ◆   func(mode); // 递归，调用自身  
  ◆ }  
  ◆ }
```

## 7.6 递归函数设计

### ——递归函数设计示例1

➤ 用辗转相除法（欧几里德算法）求两个正整数**m**和**n**的最大公约数

➤ 原理：

◆ 两个正整数的最大公约数（greatest common divisor, gcd）  
等于较小的那个数和两数相除余数的最大公约数。

◆  $\text{gcd}(m, n) = \text{gcd}(n, m \% n)$

➤ 前提/终止条件

◆ 当**m**和**n**有一个数为**0**时，另一个数就是所求的最大公约数

➤ 函数

```
◆ int gcd(int m, int n){           // 欧几里德算法(辗转相除法)
◆   if( m*n == 0 )                 // 递归终止的条件
◆     return m==0?n:m;
◆   return gcd(n, m%n);           // 递归表达式
◆ }
```

## 7.6 递归函数设计

### ——递归函数设计示例2

➤ **Fibonacci**数列，又称为黄金分割数列，指如下的数列

◆ **0, 1, 1, 2, 3, 5, 8, 13, 21, .....**

➤ 在数学上，**Fibonacci**数列被以如下的形式递归定义：

◆  **$F_0=0, F_1=1, F_n=F_{n-1}+F_{n-2} \ (n \geq 2)$**

➤ 前提/终止条件

◆  **$F_0=0, F_1=1$**

➤ 递归函数

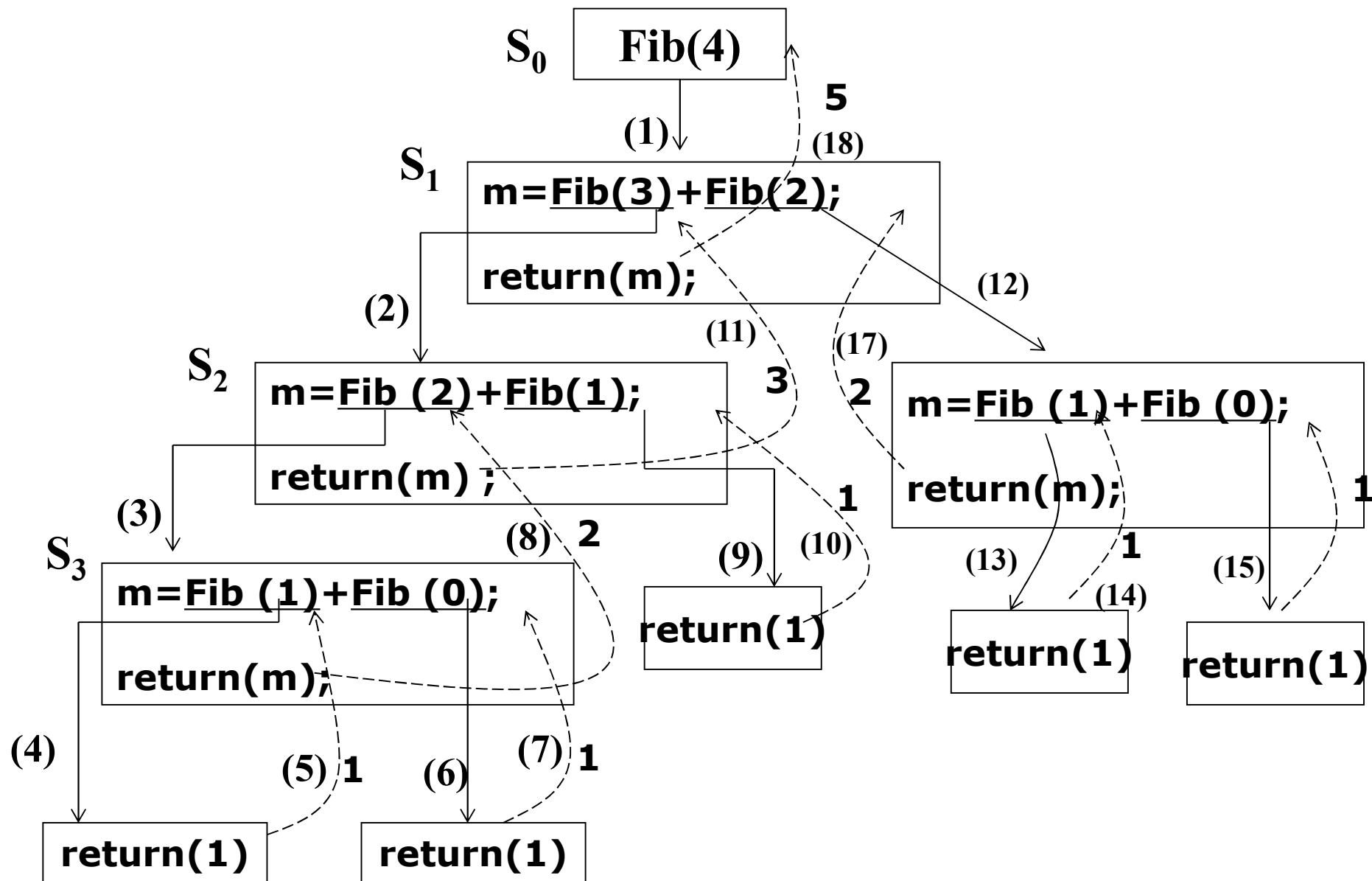
◆ **long long Fib(int n){**

◆ **if(n==0 || n==1)** //递归终止的条件

◆ **return n;**

◆ **return Fib(n-1)+Fib(n-2);** //递归表达式

◆ **}**



**Fib (4)**的执行过程

## 7.6 递归函数设计

### ——递归函数设计示例3

#### ➤ Hanoi（汉诺）塔问题

◆ 诺塔(**Tower of Hanoi**)源于印度传说中，大梵天创造世界时造了三根金钢石柱，其中一根柱子自底向上叠着**64**片黄金圆盘。大梵天命令婆罗门把圆盘从下面开始按大小顺序重新摆放在另一根柱子上。并且规定，在小圆盘上不能放大圆盘，在三根柱子之间一次只能移动一个圆盘。

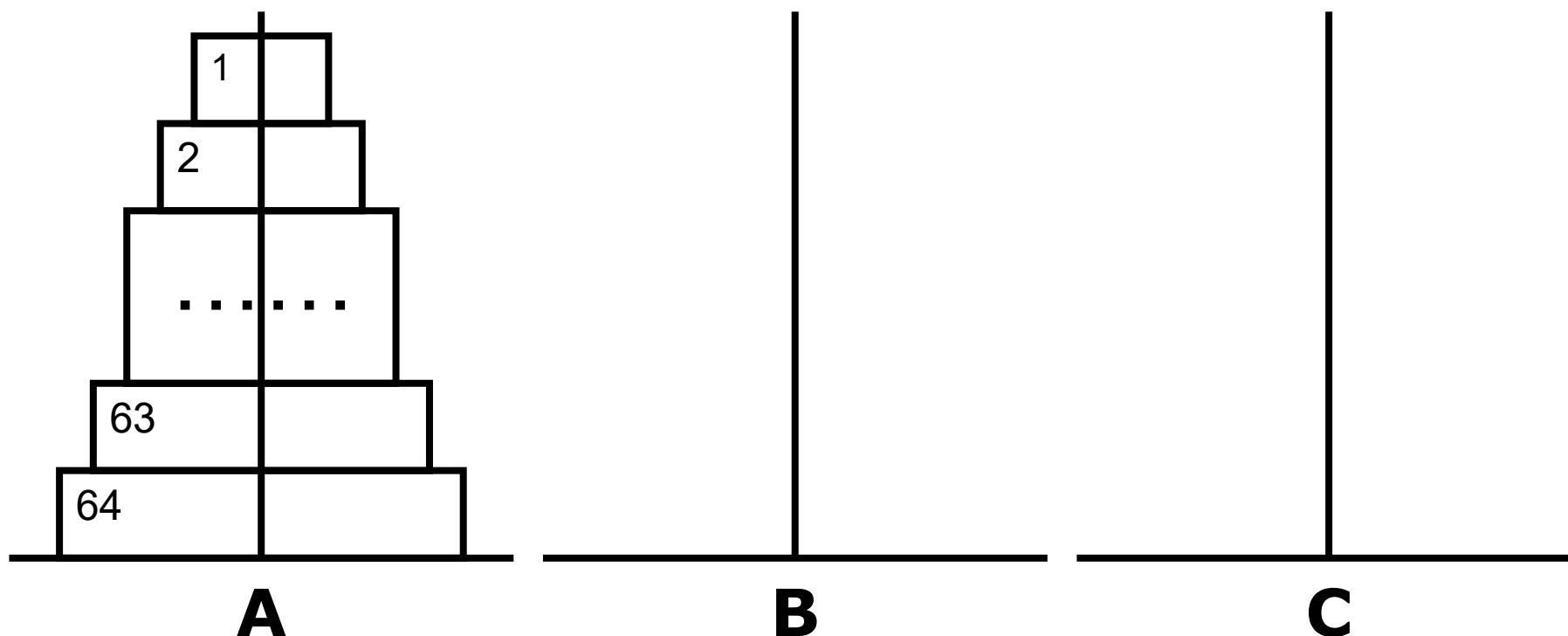
- 分别分析**1**个盘子、**2**个盘子、**3**个盘子、...的情况
- 假设有**n**片，移动次数是**f(n)**.显然**f(1)=1, f(2)=3, f(3)=7**，且 **$f(k+1) = 2 * f(k) + 1$** 。此后不难证明 **$f(n) = 2^n - 1$** ，**n=64**时，

$$f(64) = 2^{64} - 1 = 18446744073709551615$$

假如每秒钟一次，共需多长时间呢？一般一年**365**天有**31536000**秒，闰年**366**天有**31622400**秒，平均每年**31556952**秒，计算一下，

$$18446744073709551615 / 31556952 = 584554049253.855 \text{ 年} \\ \approx 5845.54 \text{ 亿年}$$



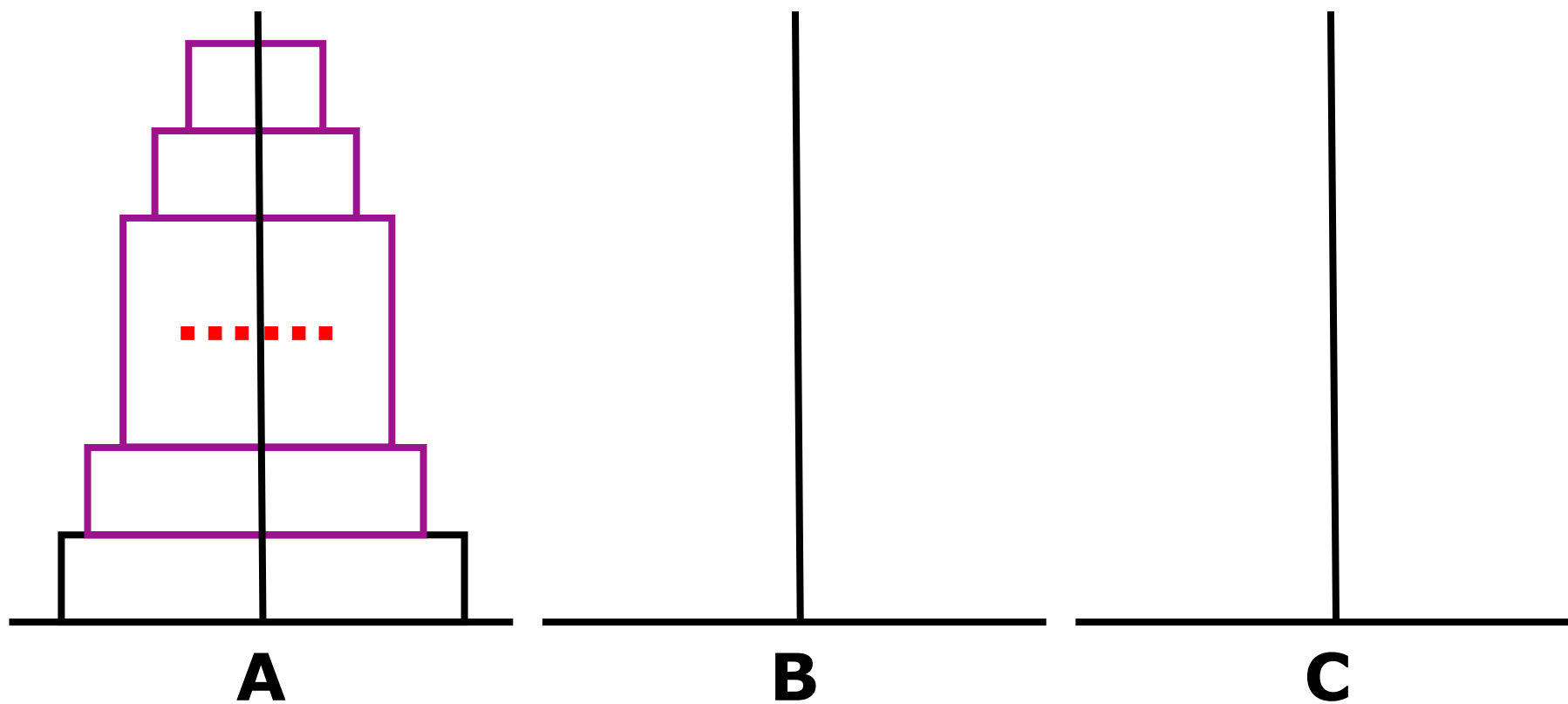


➤ 解题思路:

- ◆ 要把**64**个盘子从**A**座移动到**C**座，需要移动大约 **$2^{64}$** 次盘子。一般人是不可能直接确定移动盘子的每一个具体步骤的
- ◆ 婆罗门会这样想：假如有另外一个人能有办法将上面**63**个盘子从一个座移到另一座。那么，问题就解决了。此时婆罗门只需这样做：
  - ◆ **(1)** 命令第**2**个人将**63**个盘子从**A**座移到**B**座
  - ◆ **(2)** 自己将**1**个盘子（最底下的、最大的盘子）从**A**座移到**C**座
  - ◆ **(3)** 再命令第**2**个人将**63**个盘子从**B**座移到**C**座

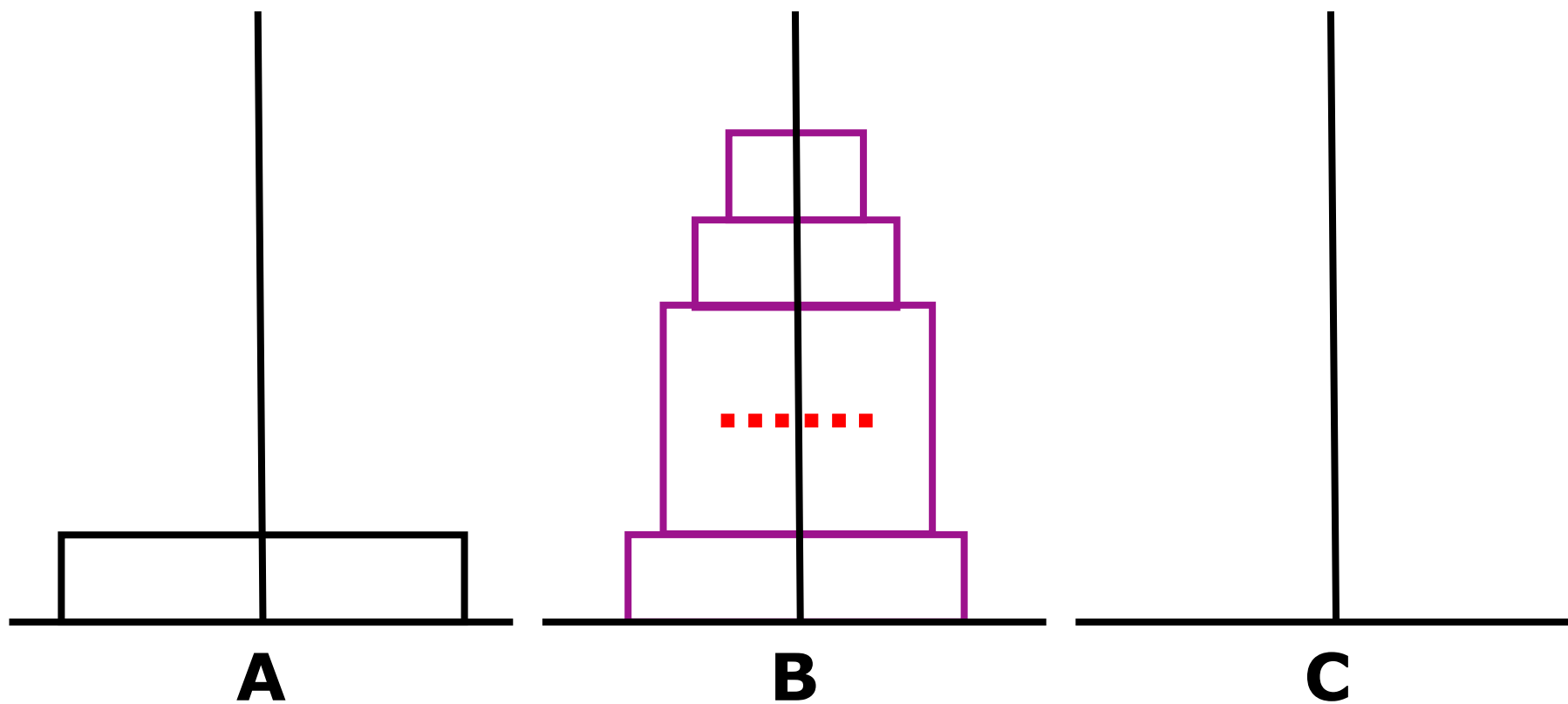
## 第**1**个人的做法

将**63**个从**A**到**B**



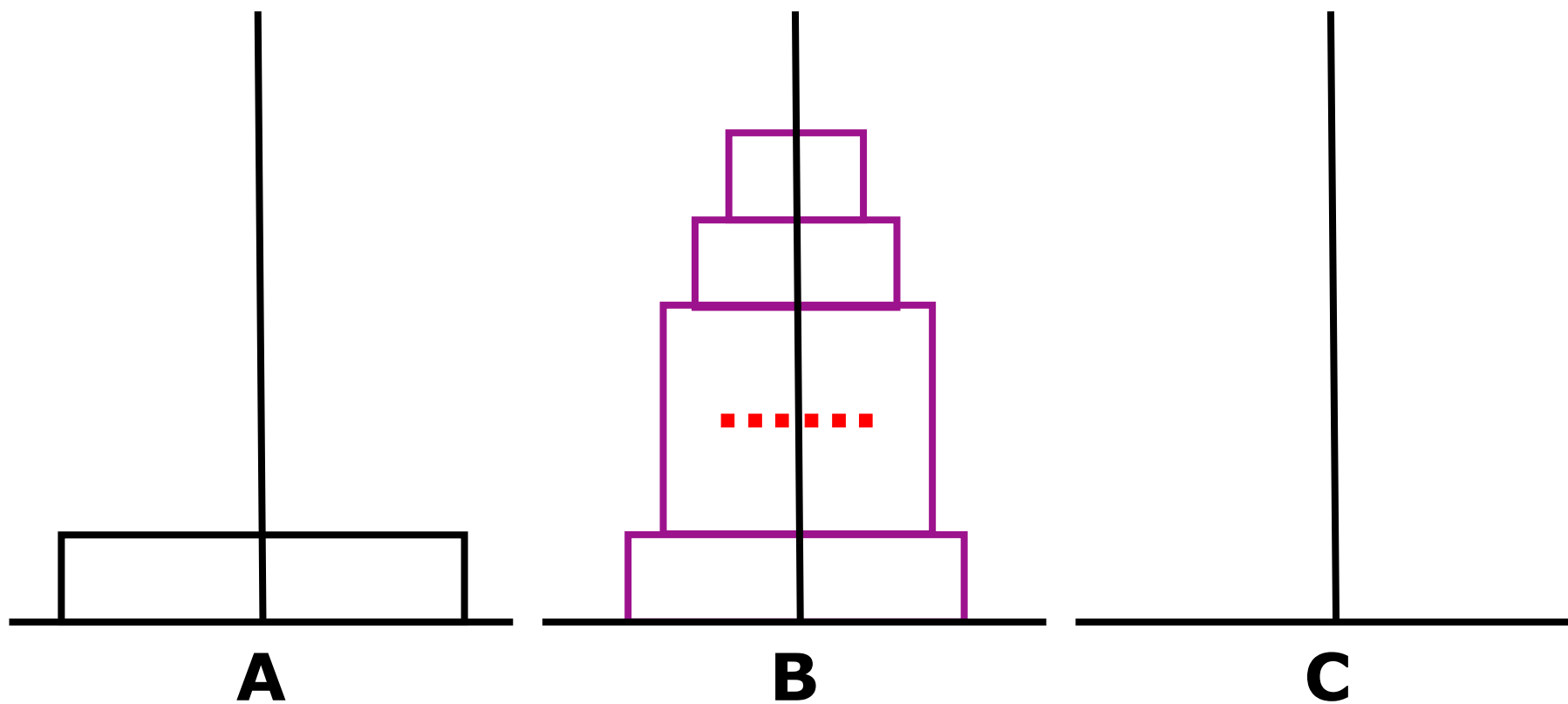
## 第**1**个人的做法

将**63**个从**A**到**B**



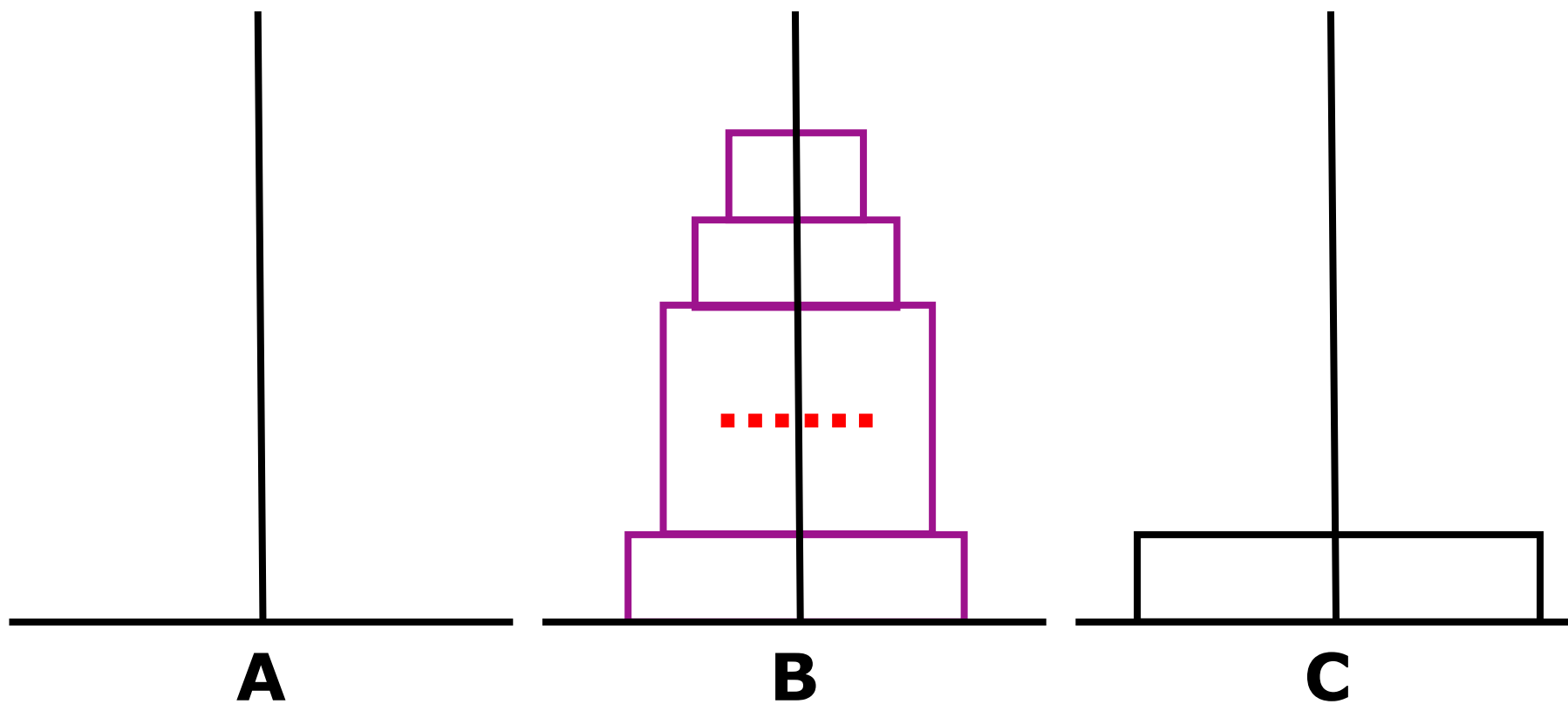
## 第**1**个人的做法

将**1**个从**A**到**C**



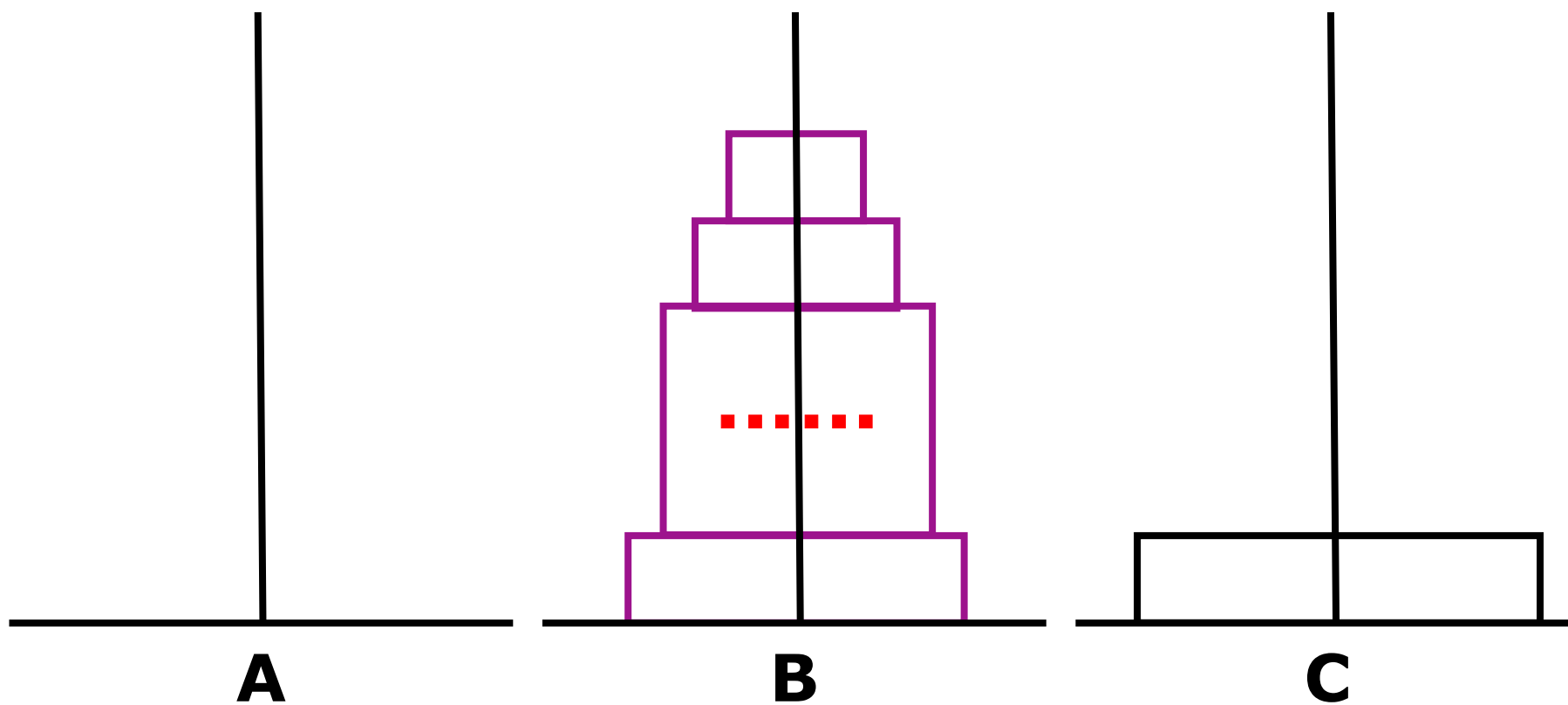
## 第**1**个人的做法

将**1**个从**A**到**C**



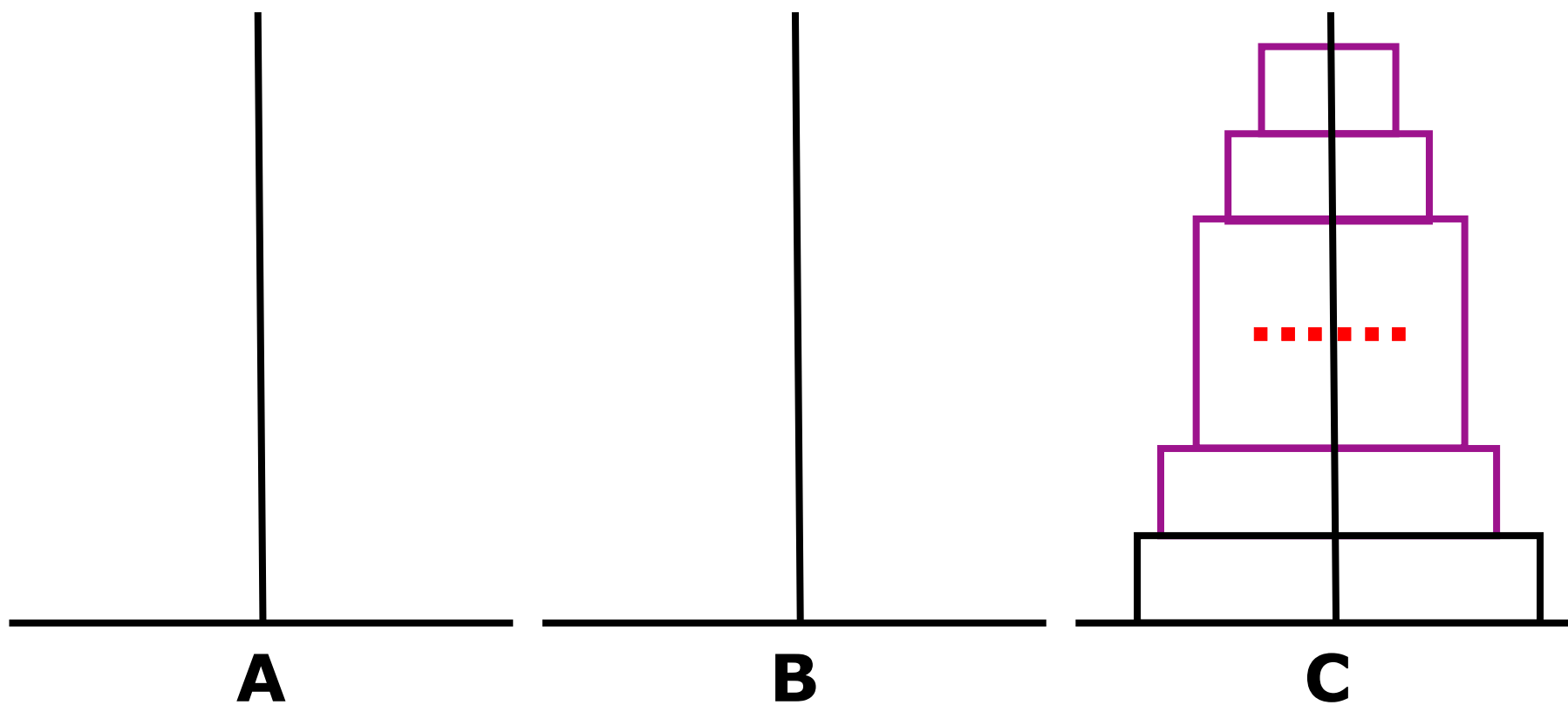
## 第**1**个人的做法

将**63**个从**B**到**C**



第**1**个人的做法

将**63**个从**B**到**C**



第**2**个人的做法

第**3**个人的做法

第**4**个人的做法

第**5**个人的做法

第**6**个人的做法

第**7**个人的做法

.....

第**63**个人的做法

第**64**个人仅做：将**1**个从**A**移到**C**



```

#include <stdio.h>
int main()
{ void hanoi(int n,char from, char mid, char to);
  int m;
  printf("the number of disks:");
  scanf("%d",&m);
  printf("move %d disks:\n",m);
  hanoi(m,'A','B','C');
}
void hanoi(int n,char from, char mid, char to)
{ void move(char from, char to);
  if(n==1)      move(from, to);
  else
  { hanoi(n-1, from, to, mid);
    move(from, to);
    hanoi(n-1, mid, from, to);
  }
}
void move(char from,char to)
{
  printf("From %c to %c\n", from, to);
}

```