数据结构与算法期末复习
Homework

尹超

中国科学院大学，北京 100049

Carter Yin

University of Chinese Academy of Sciences, Beijing 100049, China

2024.8 – 2025.1

序言

本文为笔者数据结构与算法的期末复习笔记。
望老师批评指正。

目录

题目 1-20

1

> **Hailstone 问题（又名 3n+1 问题）**：
> Hailstone(n) 的计算程序是：
> ① 对于所有的 n 都是无穷的
> ② 对于部分 n 是无穷的
> ③ 不能确定是否存在 n，使程序无法终止
> ④ 对于所有的 n 都是有穷的

Solution 1. 正确答案为 *C*。

Hailstone 问题 *(*又称 *3n+1* 问题或 *Collatz* 猜想*)* 的计算规则如下：

- 从一个正整数 *n* 开始
- 如果 *n* 是偶数，则将 *n* 除以 *2*
- 如果 *n* 是奇数，则将 *n* 乘以 *3* 再加 *1*
- 重复上述过程直到 *n=1*

例如，从 *n=3* 开始：*3→10→5→16→8→4→2→1*

目前，数学家已经验证了非常大范围内的正整数都最终会到达 *1*，但尚未能给出完整的数学证明来确定对于任意正整数该过程都会终止。这是一个著名的未解决的数学问题。

根据当前数学研究现状：

- 选项 *A* 错误：已经验证很多数字的序列最终都会终止
- 选项 *B* 错误：虽然可能存在使程序无限循环的 *n*，但尚未找到这样的例子
- 选项 *C* 正确：这是目前数学界的共识，尚无法确定是否存在使程序无法终止的 *n* 值
- 选项 *D* 错误：尚未被证明对所有 *n* 都是有穷的

因此，正确答案是选项 *C*。

2

> **判断一个算法是否是一个“好算法”，最重要的一条性质是**：
> ① 正确
> ② 健壮
> ③ 可读
> ④ 效率

Solution 2. 正确答案为 *D*。

评价一个算法的好坏，通常会考虑以下几个方面：

- **正确性 *(Correctness)***：算法能够对于每一组输入，都最终停止，并产生符合问题要求的正确输出。这是算法最基本的要求。
- **可读性 *(Readability)***：算法的逻辑清晰，易于理解、编码和调试。良好的可读性有助于算法的维护和推广。
- **健壮性 *(Robustness)***：算法能对不合法的输入做出合理的处理或反应，而不会产生异常或崩溃。

- **效率 *(Efficiency)***：算法执行时所消耗的时间和空间资源。效率通常用时间复杂度和空间复杂度来衡量。一个“好”的算法应该在满足正确性的前提下，尽可能地高效。

在这些性质中，虽然正确性是前提，但当多个算法都能正确解决问题时，**效率**往往成为衡量其优劣的最重要标准。一个低效的算法即使正确，也可能因为执行时间过长或占用资源过多而无法在实际应用中使用。

因此，正确答案是选项 *D*。

3

> **以下哪项不是图灵机的组成要件?**:
> ① 有限长的纸带
> ② 有限的字母表
> ③ 有限种状态
> ④ 读写头

Solution 3. 正确答案为 *A*。

图灵机是一种抽象的计算模型，由以下几个核心组成部分定义：

- **无限长的纸带 *(Tape)***：纸带被划分为一个个连续的单元格，每个单元格可以存储一个来自字母表的符号。理论上，这条纸带向两个方向无限延伸。
- **有限的字母表 *(Alphabet)***：一个包含有限个符号的集合，其中包括一个特殊的空白符号。这些符号可以被读写到纸带的单元格上。
- **有限种状态 *(States)***：图灵机在任何时刻都处于其有限个内部状态中的一个。其中包括一个起始状态和可能的接受/拒绝状态。
- **读写头 *(Read/Write Head)***：读写头可以在纸带上左右移动，读取当前单元格的符号，并根据当前状态和读取的符号来写入新的符号、改变内部状态以及移动读写头。
- **转移函数 *(Transition Function)***：这是一个规则集，它规定了当图灵机处于某个状态并读取到某个符号时，应该执行什么操作（写入什么符号、转换到哪个新状态、以及读写头向左还是向右移动）。

根据定义，图灵机的纸带是**无限长**的，这是其能够模拟所有可计算函数的基础。选项 *A* 描述的是“有限长的纸带”，这与图灵机的标准定义不符。

因此，正确答案是选项 *A*。

4

> **判断正误：RAM 模型与图灵机模型的区别在于图灵机的存储空间无限，而 RAM 的存储空间有限。**:
> ① 对
> ② 错

Solution 4. 正确答案为 *B (错)*。

该说法并不完全准确，是对两个模型特点的过度简化。

- **图灵机 *(Turing Machine, TM)***：其核心特征之一确实是拥有一条无限长的纸带作为存储空间。读写头在纸带上顺序移动进行操作。

- **随机存取机 *(Random Access Machine, RAM)* 模型**：*RAM* 模型假设其内存由一系列可独立寻址的存储单元（寄存器）组成。其关键特性是可以在常数时间内访问任何一个存储单元，即“随机存取”。在理论分析中，*RAM* 模型的存储空间通常也假设是无限的，或者至少是足够大的，不会成为算法分析的瓶颈（例如，可以存储与输入规模相关的多项式大小的数据）。

主要区别：

- **存储访问方式**：图灵机是顺序访问，*RAM* 模型是随机访问。
- **访问成本**：图灵机访问远距离单元格需要移动读写头，耗时与距离成正比。*RAM* 模型访问任何单元通常假设为单位时间成本。

虽然物理计算机的 *RAM* 是有限的，但在理论计算机科学中，*RAM* 模型作为一种计算模型，其存储容量通常被认为是无限的，以便与图灵机在计算能力和复杂性等级上进行比较。因此，简单地说“*RAM* 的存储空间有限”作为与图灵机无限存储的主要区别是不准确的，尤其是在讨论理论模型时。两个模型在理论上都可以拥有无限的存储资源，其根本区别在于存储结构和访问机制。

因此，原命题错误。

5

> **在大 O 记号的意义下，以下哪一项与 O(n^3) 相等？(m 不是常数)**：
> ① O(3n)
> ② O($n^3 + 2000n^2 + 1000n$)
> ③ O($n^3 + m$)
> ④ O($2000n^3 + n^4$)

Solution 5. 正确答案为 *B*。

大 *O* 记号用于描述函数渐近行为的上界。当评估一个函数的复杂度 *O(f(n))* 时，我们关注的是当输入规模 *n* 趋向于无穷大时，函数的增长率。其主要规则包括：

- **忽略低阶项**：对于多项式，只保留最高阶的项。例如，$n^3 + n^2$ 中的 n^2 是低阶项。
- **忽略常数系数**：最高阶项的常数系数被忽略。例如，$5n^3$ 中的 *5*。
- **加法规则**：*O(f(n) + g(n)) = O(max(f(n), g(n)))*。

让我们分析各个选项：

- **选项 *A: O(3n)***
  根据规则，*O(3n) = O(n)*。这与 *O(n^3)* 不相等。
- **选项 *B: O($n^3 + 2000n^2 + 1000n$)***
  在此表达式中，n^3 是最高阶项。$2000n^2$ 和 $1000n$ 是低阶项。根据加法规则和忽略低阶项及常数系数的规则，*O($n^3 + 2000n^2 + 1000n$) = O(n^3)*。这与题目要求相符。
- **选项 *C: O($n^3 + m$)***
  题目说明 *m* 不是常数。根据加法规则，*O($n^3 + m$) = O(max(n^3, m))*。如果 m 的增长阶数低于或等于 n^3 *(即 $m = O(n^3)$)*，例如 $m = n^2$ 或 $m = n^3$，那么 *O($n^3 + m$) = O(n^3)*。但是，如果 m 的增长阶数高于 n^3 *(例如，$m = n^4$)*，那么 *O($n^3 + m$) = O(m) (此时为 O(n^4))*，这就不等于 *O(n^3)*。由于题目仅说明 m 不是常数，但没有给出 m 相对于 n 的具体增长信息，我们不能保证 *O($n^3 + m$)* 总是等于 *O(n^3)*。
- **选项 *D: O($2000n^3 + n^4$)***
  在此表达式中，n^4 是最高阶项。根据加法规则和忽略低阶项及常数系数的规则，*O($2000n^3 + n^4$) = O(n^4)*。这与 *O(n^3)* 不相等。

因此，唯一明确与 $O(n^3)$ 相等的是选项 B。

6

> **下列对应关系中错误的是**:
> ① $1^2+2^2+3^2+\cdots+n^2=O(n^3)$
> ② $1+2+4+\cdots+2^n=O(2^n)$
> ③ $\log 1+\log 2+\log 3+\cdots+\log n=O(n\log n)$
> ④ $1+1/2+1/3+\cdots+1/n=O(n\log n)$

Solution 6. 正确答案为 D。

我们分析各个选项的紧确界 (Θ 记号):

- **选项 *A:*** $1^2+2^2+3^2+\cdots+n^2=O(n^3)$
  平方和公式为 $\sum_{k=1}^{n}k^2=\frac{n(n+1)(2n+1)}{6}$. 这是一个关于 n 的三次多项式，所以其数量级为 $\Theta(n^3)$。因此，$O(n^3)$ 是一个正确的紧确上界。
- **选项 *B:*** $1+2+4+\cdots+2^n=O(2^n)$
  这是一个等比数列求和 $\sum_{k=0}^{n}2^k=\frac{2^{n+1}-1}{2-1}=2^{n+1}-1$. 其数量级为 $\Theta(2^{n+1})=\Theta(2\cdot 2^n)=\Theta(2^n)$。因此，$O(2^n)$ 是一个正确的紧确上界。
- **选项 *C:*** $\log 1+\log 2+\log 3+\cdots+\log n=O(n\log n)$
  该和式等于 $\log(n!)$。根据斯特林近似公式 ($\ln(n!)\approx n\ln n-n$)，我们知道 $\log(n!)=\Theta(n\log n)$。因此，$O(n\log n)$ 是一个正确的紧确上界。
- **选项 *D:*** $1+1/2+1/3+\cdots+1/n=O(n\log n)$
  该和式是调和级数 H_n。我们知道 $H_n=\Theta(\log n)$。题目中给出的对应关系是 $H_n=O(n\log n)$。从严格的数学定义来看，这个表述是正确的，因为 $\log n\le c\cdot n\log n$ 对于足够大的 n 和某个常数 $c>0$ (例如 $c=1, n\ge 1$) 是成立的。然而，在算法分析和复杂度比较的上下文中，通常期望找到最紧确的界。H_n 的紧确界是 $O(\log n)$ (或 $\Theta(\log n)$)。与 $O(\log n)$ 相比，$O(n\log n)$ 是一个非常宽松的上界。选项 A、B、C 都给出了相应序列的紧确大 O 表示 (即它们也是 Θ 界)。如果题目旨在找出哪个对应关系没有给出最紧确或最贴切的复杂度描述，那么选项 D 符合这种情况。

因此,在评估算法复杂度时,选项 D 的表述 $O(n\log n)$ 虽然数学上是一个上界,但因为它不是紧确界 $\Theta(\log n)$,所以在这个意义下被认为是“错误”的对应关系。

7

> **判断：减而治之的思想是：将问题划分为两个平凡的子问题，分别求解子问题，来得到原问题的解。**:
> ① 对
> ② 错

Solution 7. 正确答案为 B (错)。

“减而治之”(*Decrease and Conquer*) 和“分而治之”(*Divide and Conquer*) 是两种不同的算法设计策略。

- **减而治之 (*Decrease and Conquer*)**: 这种策略通过将原问题转化为一个或多个规模更小的子问题（通常是单个子问题），然后递归地解决这些子问题，最终将子问题的解组合起来得到原问题的解。关

键在于“减”——问题规模的缩减。例如，二分查找（问题规模减半）、插入排序（待排序部分减 *1*）、计算 a^n *(*通过 $a \cdot a^{n-1}$，问题规模减 *1)* 等。

- **分而治之 *(Divide and Conquer)***：这种策略将原问题划分为若干个（通常是两个或更多）规模大致相当的、相互独立的子问题，递归地解决这些子问题，然后将子问题的解合并起来得到原问题的解。例如，归并排序、快速排序、大整数乘法等。

题目描述的“将问题划分为两个平凡的子问题，分别求解子问题，来得到原问题的解”更符合“分而治之”的思想，特别是当这两个子问题是相互独立且规模减小的时候。

减而治之的核心在于将问题规模减小，通常是减小到一个子问题。例如，计算阶乘 $n!$ 可以通过计算 $(n-1)!$ 然后乘以 n 来得到，这里就是将问题 $P(n)$ 简化为 $P(n-1)$。

因此，原命题错误。

8

> **用分而治之的思想来解决长度为 n 的数组的求和问题（n 足够大），递归实例的数目会比用减而治之的方法少。**:
>
> ① 对
>
> ② 错

Solution 8. 正确答案为 *B (*错*)*。

我们来分析两种方法解决数组求和问题时产生的递归实例数目（即递归函数的调用次数)。

- **分而治之 *(Divide and Conquer)* 求解数组求和**：该策略将数组分成两半，分别递归求这两半的和，然后将两个和相加。设 $S(A, low, high)$ 为求解数组 A 从索引 low 到 $high$ 的和的函数。

```
S(A, low, high):
    if low == high:
        return A[low]
    mid = (low + high) / 2
    sum_left = S(A, low, mid)
    sum_right = S(A, mid + 1, high)
    return sum_left + sum_right
```

对于长度为 n 的数组，如果 n 是 *2* 的幂，递归调用次数（即递归树中的节点数）为 $2n-1$。例如，对于 $n=4$，调用树有 $2*4-1=7$ 个节点。

- **减而治之 *(Decrease and Conquer)* 求解数组求和**：最常见的减而治之方法是“减一治之”(*decrease-by-one*)。设 $S(A, k)$ 为求解数组 A 前 k 个元素的和的函数。

```
S(A, k):
    if k == 0:
        return 0
    if k == 1:
        return A[0]
    return A[k-1] + S(A, k-1)
```

对于长度为 n 的数组，调用 $S(A,n)$ 会依次调用 $S(A,n-1), S(A,n-2), \ldots, S(A,1), S(A,0)$。总的调用次数为 $n+1$。

比较递归实例数目：

- 分而治之：大约 $2n-1$ 次调用。
- 减而治之（减一法）：$n+1$ 次调用。

题目陈述"分而治之的递归实例数目会比减而治之的方法少"。即判断 $2n-1 < n+1$ 是否成立。$2n-1 < n+1 \implies n < 2$。这个结论对于"*n* 足够大"的情况是不成立的。当 $n \geq 2$ 时，$2n-1 \geq n+1$。例如：

- 若 $n=1$*: 分而治之 1 次，减而治之 2 次 (S(1) -> S(0))*。此时分而治之较少。
- 若 $n=2$*: 分而治之 3 次 (S(0,1) -> S(0,0), S(1,1))*，减而治之 *3 次 (S(2) -> S(1) -> S(0))*。此时相等。
- 若 $n=3$*: 分而治之 5 次 (*近似 $2n-1$*)*，减而治之 *4 次 (*$n+1$*)*。此时分而治之较多。
- 若 $n=4$*: 分而治之 7 次 (*$2 \cdot 4 - 1 = 7$*)*，减而治之 *5 次 (*$4+1=5$*)*。此时分而治之较多。

对于足够大的 n *(*例如 $n \geq 3$*)*，分而治之的递归实例数目 *(*$2n-1$*)* **不会少于** 减而治之的递归实例数目 *(*$n+1$*)*，实际上是更多。

因此，原命题错误。

9

> **直接用定义以递归的方式计算 fib(n) 的时间复杂度是:**:
>
> ① $\Theta(n^2)$
> ② $O(2^n)$
> ③ $\Theta(2^n)$
> ④ $O(n)$

Solution 9. 正确答案为 B。

斐波那契数列的递归定义如下：

- $fib(0) = 0$
- $fib(1) = 1$
- $fib(n) = fib(n-1) + fib(n-2)$ *for* $n \geq 2$

直接使用这个定义以递归方式计算 $fib(n)$ 的伪代码如下：

```
function fib(n):
    if n <= 1:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

设 $T(n)$ 为计算 $fib(n)$ 所需的时间。

- $T(0) = \Theta(1)$
- $T(1) = \Theta(1)$
- $T(n) = T(n-1) + T(n-2) + \Theta(1)$ *for* $n \geq 2$ *(*因为有两次递归调用和一次加法操作*)*

这个递归关系式解为 $T(n) = \Theta(\phi^n)$，其中 $\phi = \frac{1+\sqrt{5}}{2} \approx 1.61803...$ *(*黄金分割比例*)*。

现在我们分析各个选项：

- **选项 *A:*** $\Theta(n^2)$
  指数函数 ϕ^n 的增长速度远快于多项式函数 n^2。因此，此选项错误。
- **选项 *B:*** $O(2^n)$
  大 O 记号表示上界。我们需要判断是否存在常数 $c > 0$ 和 n_0，使得对于所有 $n \geq n_0$，$T(n) \leq c \cdot 2^n$。由于 $T(n) = \Theta(\phi^n)$，这意味着 $T(n)$ 的增长率与 ϕ^n 同阶。因为 $\phi \approx 1.618 < 2$，所以 ϕ^n 的增长速度慢于 2^n。更准确地说，$\lim_{n\to\infty} \frac{\phi^n}{2^n} = \lim_{n\to\infty} (\frac{\phi}{2})^n = 0$，因为 $\frac{\phi}{2} < 1$。这意味着 $\phi^n = o(2^n)$ *(小 o 记号)*，同时也意味着 $\phi^n = O(2^n)$。因此，此选项正确。
- **选项 *C:*** $\Theta(2^n)$
  大 Θ 记号表示紧确界，即 $T(n) = O(2^n)$ 且 $T(n) = \Omega(2^n)$。我们已经知道 $T(n) = O(2^n)$。现在判断 $T(n) = \Omega(2^n)$ 是否成立。这要求存在常数 $c' > 0$ 和 n'_0，使得对于所有 $n \geq n'_0$，$T(n) \geq c' \cdot 2^n$。由于 $T(n) = \Theta(\phi^n)$ 且 $\phi < 2$，ϕ^n 的增长速度实际上严格慢于 2^n。因此，$T(n)$ 不是 $\Omega(2^n)$。所以，$T(n)$ 不是 $\Theta(2^n)$。此选项错误。
- **选项 *D:*** $O(n)$
  指数函数 ϕ^n 的增长速度远快于线性函数 n。因此，此选项错误。

综上所述，最合适的答案是 B，因为 $O(2^n)$ 是 $T(n) = \Theta(\phi^n)$ 的一个正确的（尽管不是最紧的，如果 ϕ^n 是选项的话）上界。

10

> **以现在普通计算机的速度，直接用定义以递归的方式计算 fib(100) 需要多少时间（不考虑溢出）:**:
> ① 一小时之内
> ② 大约一天
> ③ 十年
> ④ 这辈子看不到啦

Solution 10. 正确答案为 D。

正如在问题 *9* 中讨论的，直接用递归定义计算 $fib(n)$ 的时间复杂度是 $T(n) = \Theta(\phi^n)$，其中 $\phi = \frac{1+\sqrt{5}}{2} \approx 1.61803$ 是黄金分割比例。这意味着计算 $fib(n)$ 所需的操作次数（例如，函数调用或加法次数）大致与 ϕ^n 成正比。

对于 $fib(100)$，我们需要估算 ϕ^{100} 的大小：$\phi^{10} = (\frac{1+\sqrt{5}}{2})^{10} \approx (1.61803)^{10} \approx 122.88$ $\phi^{100} = (\phi^{10})^{10} \approx (122.88)^{10}$

我们可以用对数来更精确地估算：$\log_{10}(\phi) \approx \log_{10}(1.61803) \approx 0.208987$ $\log_{10}(\phi^{100}) = 100 \times \log_{10}(\phi) \approx 100 \times 0.208987 = 20.8987$ 所以，$\phi^{100} \approx 10^{20.8987} = 10^{0.8987} \times 10^{20}$. $10^{0.8987} \approx 7.919$. 因此，$\phi^{100} \approx 7.919 \times 10^{20}$.

这意味着计算 $fib(100)$ 大约需要 $k \cdot (7.919 \times 10^{20})$ 次基本操作，其中 k 是一个小的常数。即使我们假设 $k = 1$，这也是一个巨大的数字。

假设一台“普通计算机”每秒可以执行 10^9 次操作 *(1 Giga-operations per second, or 1 GHz CPU where each cycle performs one relevant operation, which is optimistic).* 所需时间 *(秒)* $\approx \frac{7.919 \times 10^{20} \text{ operations}}{10^9 \text{ operations/second}} = 7.919 \times 10^{11}$ 秒。

现在将这个时间转换为更易于理解的单位：

- *1* 分钟 = *60* 秒
- *1* 小时 = $60 \times 60 = 3600$ 秒
- *1* 天 = $24 \times 3600 = 86400$ 秒
- *1* 年 $\approx 365.25 \times 86400 \approx 3.15576 \times 10^7$ 秒

所需时间 *(年)* $\approx \frac{7.919\times10^{11}\ 秒}{3.15576\times10^{7}\ 秒/年} \approx 2.509 \times 10^4$ 年。

2.509×10^4 年等于 *25,090* 年。

这个时间远远超过了一个人的一生，甚至人类有记载的历史。

- 选项 *A (一小时之内)*：3600 秒。$7.919 \times 10^{11} \gg 3600$. 错误。
- 选项 *B (大约一天)*：86400 秒。$7.919 \times 10^{11} \gg 86400$. 错误。
- 选项 *C (十年)*：$10 \times 3.15576 \times 10^7 \approx 3.15 \times 10^8$ 秒。$7.919 \times 10^{11} \gg 3.15 \times 10^8$. 错误。
- 选项 *D (这辈子看不到啦)*：*25,090* 年显然是“这辈子看不到啦”。正确。

因此，由于递归计算斐波那契数的指数级时间复杂度，计算 $fib(100)$ 所需的时间是天文数字。

11

> **用动态规划计算 fib(n) 的时间、空间复杂度分别为:**:
> ① $\Theta(n^2), \Theta(n^2)$
> ② $\Theta(n^2), \Theta(n)$
> ③ $\Theta(n), \Theta(n)$
> ④ $\Theta(n), \Theta(1)$

Solution 11. 正确答案为 *D*。

动态规划 *(Dynamic Programming, DP)* 的核心思想是存储已解决的子问题的结果，以避免重复计算。对于斐波那契数列 $fib(n) = fib(n-1) + fib(n-2)$ *(*其中 $fib(0) = 0, fib(1) = 1$*)*，有两种主要的动态规划实现方式:

***1.* 自底向上 *(Bottom-Up)* 使用数组存储:** 这种方法创建一个数组（例如 *'dp'*）来存储从 $fib(0)$ 到 $fib(n)$ 的所有斐波那契数。

- *'dp[0] = 0'*
- *'dp[1] = 1'*
- *'for i from 2 to n:'*
- *' dp[i] = dp[i-1] + dp[i-2]'*
- *'return dp[n]'*
- **时间复杂度**：循环从 *2* 到 *n* 执行，共 $n-1$ 次。每次循环内部的操作（两次数组读取和一次加法）都是常数时间 $\Theta(1)$。因此，总时间复杂度为 $\Theta(n)$。
- **空间复杂度**：需要一个大小为 $n+1$ 的数组来存储所有斐波那契数。因此，空间复杂度为 $\Theta(n)$。

这种方法对应选项 *C*：$\Theta(n), \Theta(n)$。

***2.* 自底向上 *(Bottom-Up)* 空间优化:** 观察到计算 $fib(i)$ 只需要前两个斐波那契数 $fib(i-1)$ 和 $fib(i-2)$。因此，我们不需要存储整个数组，只需要保留最近的两个值。

- `if n == 0: return 0`
- `if n == 1: return 1`
- `a = 0` *(*$fib(0)$*)*
- `b = 1` *(*$fib(1)$*)*
- `for i from 2 to n:`
- `  current_fib = a + b`
- `  a = b`
- `  b = current_fib`

- `return b`
- **时间复杂度**：循环从 *2* 到 *n* 执行，共 $n-1$ 次。每次循环内部的操作都是常数时间 $\Theta(1)$。因此，总时间复杂度为 $\Theta(n)$。
- **空间复杂度**：只需要固定数量的变量（例如 *a, b, current_fib, i*）来存储中间结果，与 n 的大小无关。因此，空间复杂度为 $\Theta(1)$。

这种方法对应选项 *D*：$\Theta(n), \Theta(1)$。

结论：两种方法都是动态规划的有效实现。空间优化的方法在时间复杂度相同的情况下，空间效率更高。题目问的是“用动态规划计算”，通常会考虑其最优或常见的实现。

- 选项 *A* 和 *B* 的时间复杂度为 $\Theta(n^2)$，这对于 *DP* 解斐波那契数列是不正确的。
- 选项 *C (*$\Theta(n), \Theta(n)$*)* 描述了使用数组的 *DP* 方法。
- 选项 *D (*$\Theta(n), \Theta(1)$*)* 描述了空间优化的 *DP* 方法。

由于空间优化的 *DP* 方法是计算斐波那契数列的标准高效 *DP* 解法，并且选项 *D* 给出了这种方法的复杂度，因此选项 *D* 是最佳答案。