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 第二章

12

> **下列说法中正确的是:**:
>
> ① 抽象数据类型是 C++ 提供的一种高级的数据类型，用于实现数据结构。
>
> ② 抽象数据类型决定了数据的存储方式。
>
> ③ 同一个抽象数据类型可能用多种数据结构实现。
>
> ④ 数据结构即抽象数据类型。

Solution 1. 正确答案为 *C*。

- **抽象数据类型 *(Abstract Data Type, ADT)***：*ADT* 是一个数学模型，它定义了一组数据以及在该组数据上的一组操作。它关注的是数据的逻辑特性和在其上可以执行的操作，而不关心这些数据如何存储或这些操作如何实现。*ADT* 强调的是“是什么”*(what)*，即数据对象以及可以对这些对象执行的操作的规范。
- **数据结构 *(Data Structure)***：数据结构是计算机中存储、组织数据的方式。它是 *ADT* 的一种具体实现。数据结构关注的是“如何做”*(how)*，即如何在计算机内存中表示数据以及如何实现 *ADT* 定义的操作。

分析各个选项：

- **选项 *A:* 抽象数据类型是 *C++* 提供的一种高级的数据类型，用于实现数据结构。**
  错误。抽象数据类型是一个理论概念，而不是特定编程语言（如 *C++*）直接提供的数据类型。*C++* 提供了类 *(class)* 等机制来帮助程序员实现 *ADT*。数据结构是用来实现 *ADT* 的，而不是反过来。
- **选项 *B:* 抽象数据类型决定了数据的存储方式。**
  错误。*ADT* 只定义了数据的逻辑视图和操作，它隐藏了内部的存储细节。数据的具体存储方式是由实现该 *ADT* 的数据结构决定的。
- **选项 *C:* 同一个抽象数据类型可能用多种数据结构实现。**
  正确。这是 *ADT* 的一个重要特性。例如，一个“列表”*ADT* 可以通过数组（顺序表）实现，也可以通过链表实现。这两种数据结构在性能特征（如插入、删除、访问时间）上可能有所不同，但它们都可以满足列表 *ADT* 定义的操作。
- **选项 *D:* 数据结构即抽象数据类型。**
  错误。数据结构是 *ADT* 的物理实现，而 *ADT* 是数据结构的逻辑描述。它们是两个不同层面但紧密相关的概念。可以将 *ADT* 视为接口，数据结构视为该接口的实现。

因此，正确的说法是选项 *C*。

13

> **在一个初始为空的向量上依次执行：insert(0, 2), insert(1, 6), put(0, 1), remove(1), insert(0, 7) 后的结果是:**:
>
> ① {6, 2, 7}
>
> ② {2, 6, 0, 7}
>
> ③ {7, 1}
>
> ④ {2, 1, 7}

Solution 2. 正确答案为 C。

我们逐步追踪向量在每次操作后的状态。假设向量的行为类似于动态数组（如 *C++* 的 *'std::vector'* 或 *Python* 的 *'list'*）。

(1) 初始状态：向量 V 为空。$V = \{\}$

(2) insert(0, 2)：在索引 *0* 处插入元素 *2*。$V = \{2\}$

(3) insert(1, 6)：在索引 *1* 处插入元素 *6*。向量中的元素会向后移动以腾出空间（如果需要，但这里是追加）。$V = \{2, 6\}$

(4) put(0, 1)：将索引 *0* 处的元素替换为 *1*。$V = \{1, 6\}$

(5) remove(1)：移除索引 *1* 处的元素（即 *6*）。移除后，后续元素（如果有）会向前移动。$V = \{1\}$

(6) insert(0, 7)：在索引 *0* 处插入元素 *7*。原索引 *0* 处的元素（即 *1*）向后移动到索引 *1*。$V = \{7, 1\}$

因此，经过所有操作后，向量的内容为 $\{7, 1\}$。

对照选项：

- *A: {6, 2, 7}* - 错误
- *B: {2, 6, 0, 7}* - 错误
- *C: {7, 1}* - 正确
- *D: {2, 1, 7}* - 错误

14

> **在一个初始最大容量为 10 的空向量上依次执行：insert(0, 2), insert(1, 6), put(0, 1), remove(1), insert(0, 7) 后的装填因子是:**:
>
> ① 10%
> ② 20%
> ③ 30%
> ④ 40%

Solution 3. 正确答案为 B。

装填因子 *(Load Factor)* 通常定义为：

$$Load\ Factor = \frac{Number\ of\ elements\ currently\ in\ the\ vector}{Current\ maximum\ capacity\ of\ the\ vector}$$

我们首先需要确定在执行完所有操作后向量中元素的数量。这些操作与问题 *13* 中的操作相同。在问题 *13* 中，我们追踪了操作过程：

(1) 初始状态：$V = \{\}$ *(0 个元素)*

(2) insert(0, 2)：$V = \{2\}$ *(1 个元素)*

(3) insert(1, 6)：$V = \{2, 6\}$ *(2 个元素)*

(4) put(0, 1)：$V = \{1, 6\}$ *(2 个元素)*

(5) remove(1)：$V = \{1\}$ *(1 个元素)*

(6) insert(0, 7)：$V = \{7, 1\}$ *(2 个元素)*

经过所有操作后，向量中包含 *2* 个元素。

题目中给出向量的初始最大容量为 *10*。我们假设在这些操作过程中，向量的容量没有因为缩容或扩容策略而改变（题目没有提供此类信息，通常在不触发扩容的情况下，容量保持不变）。

因此：

- *Number of elements = 2*
- *Maximum capacity = 10*

装填因子 = $\frac{2}{10} = 0.2$。

将其转换为百分比：$0.2 \times 100\% = 20\%$。

对照选项：

- *A: 10%* - 错误
- *B: 20%* - 正确
- *C: 30%* - 错误
- *D: 40%* - 错误

15

> **以下代码是向量复制代码的一个变体且语义与其相同，**
>
> ```
> void copyFrom(const T*A, Rank lo, Rank hi)
> {
>     _elem = new T[_capacity = 2 * (hi - lo)];
>     _size = hi -lo;
>     for(int i = _size - 1; -1 < i; i--)
>     {
>         _elem[i] = A[ ____ ];
>     }
> }
> ```
>
> **空格处应填入的内容为：**：
>
> ① –hi
> ② hi–
> ③ ++lo

Solution 4. 正确答案为 *A*。

该函数 *copyFrom* 的目的是将源数组 *A* 中从索引 *lo (包含)* 到 *hi (不包含)* 的元素复制到向量的内部存储 *_elem* 中。向量的大小 *_size* 被设置为 *hi - lo*。循环 *'for(int i = _size - 1; -1 < i; i–)'* 从 *'_size - 1'* 向下迭代到 *'0'*。这意味着：

- *_elem[_size - 1] (向量的最后一个元素)* 应该接收源数组段的最后一个元素，即 *A[hi - 1]*。
- *_elem[_size - 2]* 应该接收 *A[hi - 2]*。
- ...
- *_elem[0] (向量的第一个元素)* 应该接收源数组段的第一个元素，即 *A[lo]*。

设 *'hi_initial'* 和 *'lo_initial'* 分别是传入参数 *'hi'* 和 *'lo'* 的初始值。*'_size = hi_initial - lo_initial'*。

我们来分析各个选项：

- **选项 *A: –hi*** 在循环的每次迭代中，*'–hi'* 会先将 *'hi'* 的值减 *1*，然后使用这个新值作为数组 *'A'* 的索引。
  - ***(1) 第一次迭代 (*** $i = \_size - 1$***):*** *'hi'* 的值变为 *'hi_initial - 1'*。*'_elem[_size - 1] = A[hi_initial - 1]'*。这是正确的，向量的最后一个元素接收了源片段的最后一个元素。

(2) 第二次迭代 (*$i = \_size - 2$*): 此时 *'hi'* 的值是 *'hi_initial - 1'*。执行 *'--hi'* 后，*'hi'* 变为 *'hi_initial - 2'*。*'_elem[_size - 2] = A[hi_initial - 2]'*。这是正确的。

(3) ...

(4) 第 *k* 次迭代 (从后往前数，*i = _size - k*): 在这次迭代开始前，*'hi'* 的值是 *'hi_initial - (k-1)'*。执行 *'--hi'* 后，*'hi'* 变为 *'hi_initial - k'*。*'_elem[_size - k] = A[hi_initial - k]'*。

(5) 最后一次迭代 (*$i = 0$*): 这是第 *'_size'* 次迭代 *(k = '_size')*。在这次迭代开始前，*'hi'* 的值是 *'hi_initial - (_size - 1)'*。执行 *'--hi'* 后，*'hi'* 变为 *'hi_initial - _size'*。*'_elem[0] = A[hi_initial - _size]'*。因为 *'_size = hi_initial - lo_initial'*，所以 *'hi_initial - _size = hi_initial - (hi_initial - lo_initial) = lo_initial'*。因此，*'_elem[0] = A[lo_initial]'*。这是正确的，向量的第一个元素接收了源片段的第一个元素。

所以，选项 *A ('--hi')* 是正确的。

- **选项 *B: hi--*** *'hi--'* 会先使用 *'hi'* 的当前值作为索引，然后再将 *'hi'* 减 *1*。

  (1) 第一次迭代 (*$i = \_size - 1$*): 使用 *'hi_initial'* 作为索引。*'_elem[_size - 1] = A[hi_initial]'*。这是错误的，因为源数组段的有效索引是到 *'hi_initial - 1'*。

  所以，选项 *B* 错误。

- **选项 *C: ++lo*** *'++lo'* 会先将 *'lo'* 的值加 *1*，然后使用这个新值作为索引。

  (1) 第一次迭代 (*$i = \_size - 1$*): *'lo'* 的值变为 *'lo_initial + 1'*。*'_elem[_size - 1] = A[lo_initial + 1]'*。这通常是不正确的，因为我们期望 *'_elem[_size - 1]* 得到 *'A[hi_initial - 1]'*。

  所以，选项 *C* 错误。

因此，正确的填空是 *'--hi'*。

16

> **采用每次追加固定内存空间的扩容策略，规模为 n 的向量插入元素的分摊时间复杂度为:**:
>
> ① $\Theta(n \log_2 n)$
>
> ② $\Theta(n)$
>
> ③ $\Theta(\log_2 n)$
>
> ④ $\Theta(1)$

Solution 5. 正确答案为 *B*。

分摊时间复杂度 *(Amortized Time Complexity)* 分析的是在一系列操作中，单个操作的平均成本。

扩容策略：每次追加固定内存空间 假设向量初始容量为 C_0，每次扩容时，容量增加一个固定的量 K。当向量已满（包含 m 个元素，容量也为 m）并且需要插入一个新元素时，会发生以下情况:

(1) 分配一块新的内存，大小为 $m + K$。

(2) 将原来的 m 个元素从旧内存复制到新内存。此操作耗时 $\Theta(m)$。

(3) 插入新元素。此操作耗时 $\Theta(1)$ *(在新空间中)*。

(4) 释放旧内存。

一次扩容操作的成本主要由复制元素的成本决定，即 $\Theta(m)$，其中 m 是扩容前的元素数量。

分析一系列插入操作的成本 考虑从一个空向量开始，连续插入 n 个元素。

- 每次插入操作本身（不考虑扩容）的成本是 $\Theta(1)$。总共 n 次插入，这部分成本是 $\Theta(n)$。
- 扩容操作：假设初始容量很小或为 *0*。第一次扩容发生在插入第 $C_0 + 1$ 个元素时（如果 C_0 是初始容量），或者更简单地，假设第一次扩容后容量为 K_1 *(*可能是 K 或 $C_0 + K$*)*，复制了 $K_1 - K$ 个元

素。第二次扩容发生在元素数量达到 K_1 时，需要复制 K_1 个元素，新容量变为 K_1+K。第三次扩容发生在元素数量达到 K_1+K 时，需要复制 K_1+K 个元素，新容量变为 K_1+2K。... 第 j-次扩容（大约）发生在元素数量达到 $S_j \approx (j-1)K + K_0'$ *(某个初始有效容量)* 时，需要复制 S_j 个元素。为了插入 n 个元素，扩容会发生大约 n/K 次。在第 j-次扩容时（当元素数量大约为 jK 时），复制成本为 $\Theta(jK)$。总的复制成本为：

$$\sum_{j=1}^{\approx n/K} \Theta(jK) = \Theta(K) \sum_{j=1}^{n/K} j = \Theta(K) \cdot \Theta\left(\left(\frac{n}{K}\right)^2\right) = \Theta(K) \cdot \frac{n^2}{K^2} = \Theta\left(\frac{n^2}{K}\right)$$

由于 K 是一个固定的常数，总的复制成本是 $\Theta(n^2)$。

计算分摊成本 总成本 = （n 次基本插入的成本）+（所有扩容的复制成本）总成本 = $\Theta(n) + \Theta(n^2)$

分摊成本 = $\frac{\text{总成本}}{\text{操作次数}} = \frac{\Theta(n^2)}{n} = \Theta(n)$。

因此，采用每次追加固定内存空间的扩容策略，在向量上插入元素的平均分摊时间复杂度为 $\Theta(n)$。这与几何级数扩容（例如，每次将容量加倍）策略形成对比，后者的分摊时间复杂度为 $\Theta(1)$。

对照选项：

- *A:* $\Theta(n \log_2 n)$ - 错误
- *B:* $\Theta(n)$ - 正确
- *C:* $\Theta(\log_2 n)$ - 错误
- *D:* $\Theta(1)$ - 错误 *(这是几何扩容策略的分摊成本)*

17

> **分别采用每次追加固定内存空间和每次内存空间翻倍两种扩容策略，规模为 n 的向量插入元素的分摊时间复杂度分别为：**：
>
> ① $\Theta(n), \Theta(1)$
> ② $\Theta(n), \Theta(n)$
> ③ $\Theta(1), \Theta(1)$
> ④ $\Theta(n), \Theta(\log_2 n)$

Solution 6. 正确答案为 A。

我们分别分析两种扩容策略下的分摊时间复杂度。

1. 每次追加固定内存空间 (Additive Increase) 正如在问题 *16* 中详细分析的：

- 当向量满时，容量增加一个固定的量 K.
- 一次扩容操作（复制 m 个元素）的成本为 $\Theta(m)$.
- 为了插入 n 个元素，总的扩容成本（主要是复制成本）为 $\Theta(n^2/K) = \Theta(n^2)$ *(因为 K 是常数)*。
- n 次基本插入操作的成本为 $\Theta(n)$.
- 总成本为 $\Theta(n) + \Theta(n^2) = \Theta(n^2)$.
- 分摊时间复杂度 = $\frac{\text{总成本}}{\text{操作次数}} = \frac{\Theta(n^2)}{n} = \Theta(n)$.

所以，采用每次追加固定内存空间的扩容策略，分摊时间复杂度为 $\Theta(n)$。

2. 每次内存空间翻倍 (Multiplicative Increase / Geometric Expansion) 假设向量初始容量为 C_0 *(或从 1 开始)*。当向量已满（包含 m 个元素，容量也为 m）并且需要插入一个新元素时，会发生以下情况：

(1) 分配一块新的内存，大小为 $2m$ *(即容量翻倍)*。

(2) 将原来的 m 个元素从旧内存复制到新内存。此操作耗时 $\Theta(m)$。

(3) 插入新元素。此操作耗时 $\Theta(1)$ *(在新空间中)*。

(4) 释放旧内存。

一次扩容操作的成本主要由复制元素的成本决定，即 $\Theta(m)$。

考虑从一个空向量开始，连续插入 n 个元素。

- 每次插入操作本身（不考虑扩容）的成本是 $\Theta(1)$。总共 n 次插入，这部分成本是 $\Theta(n)$。
- 扩容操作：扩容发生在元素数量达到当前容量时。假设容量从 1 开始，依次变为 $1, 2, 4, 8, \ldots, 2^k$。当容量从 2^i 变为 2^{i+1} 时，需要复制 2^i 个元素。为了插入 n 个元素，最后一次扩容可能将容量扩展到 $2^k \geq n$。总的复制成本是 $1 + 2 + 4 + \cdots + 2^{k-1}$ *(如果 2^k 是刚好大于或等于 n 的容量，那么上一次复制的元素数量是 2^{k-1})*。这个几何级数的和是 $2^k - 1$. 由于 $2^{k-1} < n \leq 2^k$，那么 $2^k < 2n$. 所以，总的复制成本为 $\Theta(2^k) = \Theta(n)$.

计算分摊成本 总成本 = （n 次基本插入的成本）+（所有扩容的复制成本）总成本 = $\Theta(n) + \Theta(n)$

分摊成本 = $\frac{\text{总成本}}{\text{操作次数}} = \frac{\Theta(n)}{n} = \Theta(1)$。

所以，采用每次内存空间翻倍的扩容策略，分摊时间复杂度为 $\Theta(1)$。

总结：

- 每次追加固定内存空间：分摊时间复杂度 $\Theta(n)$。
- 每次内存空间翻倍：分摊时间复杂度 $\Theta(1)$。

这对应于选项 A。

对照选项：

- *A:* $\Theta(n), \Theta(1)$ - 正确
- *B:* $\Theta(n), \Theta(n)$ - 错误
- *C:* $\Theta(1), \Theta(1)$ - 错误
- *D:* $\Theta(n), \Theta(\log_2 n)$ - 错误

18

> **关于平均复杂度和分摊复杂度，下列说法中错误的是:**:
>
> ① 分摊复杂度所考量的一串操作序列一定是真实可行的
>
> ② 平均复杂度依赖于对各操作出现概率的假设，而分摊复杂度则不是如此
>
> ③ 分摊复杂度得到的结果比平均复杂度低
>
> ④ 加倍扩容策略中 $\Theta(1)$ 的结论是指分摊复杂度

Solution 7. 正确答案为 C。

- **平均复杂度 *(Average-case Complexity)***：分析算法在所有可能输入下的期望运行时间。这通常需要对输入的概率分布做出假设。例如，快速排序的平均时间复杂度是 $\Theta(n \log n)$，这是基于输入数据的所有排列等概率出现的假设。
- **分摊复杂度 *(Amortized Complexity)***：分析在一系列操作中，单个操作的平均成本。它考虑的是最坏情况下的操作序列，并计算该序列中每个操作的平均成本。分摊分析不依赖于概率假设，它保证了在任何操作序列中，平均每个操作的成本不会超过某个界限。

现在分析各个选项：

- **选项 *A:* 分摊复杂度所考量的一串操作序列一定是真实可行的**

  正确。分摊分析是对数据结构执行一系列实际操作的成本进行平均。这些操作构成了在数据结构上可能发生的真实序列。

- **选项 *B:* 平均复杂度依赖于对各操作出现概率的假设，而分摊复杂度则不是如此**
  正确。这是两者之间的一个关键区别。平均复杂度需要一个关于输入或操作序列的概率模型，而分摊复杂度提供的是在任何可能的操作序列（即最坏情况序列）上的平均性能保证，无需概率假设。
- **选项 *C:* 分摊复杂度得到的结果比平均复杂度低**
  错误。这个说法不总是成立。分摊复杂度和平均复杂度是两种不同的衡量标准，它们之间没有固定的"谁高谁低"的关系。
  - 分摊复杂度分析的是最坏情况序列中的平均操作成本。
  - 平均复杂度分析的是所有可能输入的期望成本，基于概率分布。

  如果一个操作的"最坏情况序列"在平均情况分析的概率模型中非常罕见，那么平均复杂度可能会远低于分摊复杂度。反之，如果概率模型使得导致较高成本的操作频繁出现（即使不是严格的"最坏序列"），平均复杂度也可能不低。例如，如果一个操作通常成本很低，但在极罕见的情况下成本非常高，其平均复杂度可能很低。其分摊复杂度则取决于这种高成本是否能被序列中其他操作"平摊"掉。因此，不能一概而论分摊复杂度一定比平均复杂度低。
- **选项 *D:* 加倍扩容策略中 $\Theta(1)$ 的结论是指分摊复杂度**
  正确。正如在问题 *17* 中讨论的，当动态数组（向量）采用容量加倍的扩容策略时，虽然单次扩容操作可能耗时 $\Theta(m)$（其中 m 是当前元素数量），但在一系列插入操作中，平均每次插入的分摊时间复杂度是 $\Theta(1)$。这是分摊分析的一个经典例子。

因此，错误的说法是选项 *C*。

19

> **向量 disordered() 算法的返回值是:**:
> ① 逆序数
> ② 相邻逆序对个数
> ③ 表示是否有序的 bool 值
> ④ 表示是否有序的 int 值

Solution 8. 正确答案为 *B*。

向量 *'disordered()'* 算法通常用于检查向量是否有序，或者量化其无序程度。在常见的实现中（例如，在一些数据结构教材或库的上下文中），*'disordered()'* 方法会返回相邻元素逆序对的个数。

一个典型的实现如下:

```
template <typename T>
int Vector<T>::disordered() const { // 返回向量中逆序相邻元素对的总数
    int n = 0; //计数器
    for (int i = 1; i < _size; i++) { //逐一检查_size-1对相邻元素
        if (_elem[i-1] > _elem[i]) { //若该对元素逆序，则
            n++; //计数器自增
        }
    }
    return n; //返回逆序对的总数；仅当n为0时，向量有序
}
```

分析这个实现和各个选项：

- **返回值**：该函数返回一个整数 `n`。
- **`n` 的含义**：`n` 统计的是满足 `_elem[i-1] > _elem[i]` 条件的相邻元素对的数量。这正是“相邻逆序对个数”。

对照选项：

- **选项 *A: 逆序数***
  错误。总的逆序数是指所有满足 $j < k$ 但 $A[j] > A[k]$ 的元素对 $(A[j], A[k])$ 的数量。计算总逆序数通常比计算相邻逆序对个数更复杂。*'disordered()'* 的简单实现一般不计算这个。
- **选项 *B:* 相邻逆序对个数**
  正确。如上述典型实现所示，*'disordered()'* 函数直接计算并返回相邻元素中前者大于后者（即构成逆序）的对数。
- **选项 *C:* 表示是否有序的 *bool* 值**
  错误。虽然可以通过检查 `disordered()` 的返回值是否为 *0* 来判断向量是否有序（返回 *0* 则有序，非 *0* 则无序），但函数本身返回的是一个计数值（*int*），而不是直接的布尔值。如果函数旨在返回布尔值，它通常会命名为类似 `is_sorted()`。
- **选项 *D:* 表示是否有序的 *int* 值**
  部分正确但不够精确。函数确实返回一个 *int* 值，并且这个 *int* 值可以用来判断是否有序（*0* 表示有序）。然而，选项 *B* 更准确地描述了这个 *int* 值的具体含义——它不仅仅是一个标志，而是相邻逆序对的实际数量。

因此，最准确的描述是 *'disordered()'* 算法返回相邻逆序对的个数。

20

> **为什么有序向量唯一化算法中不需要调用 remove() 进行元素删除?**:
> ① 本来就没有重复元素
> ② 重复元素被直接忽略了
> ③ 重复元素被移到了向量末尾
> ④ 重复元素修改成了不重复的元素

Solution 9. 正确答案为 *B*。

有序向量的唯一化（*uniquify*）算法旨在高效地移除重复元素，使得每个元素只出现一次，同时保持原有元素的相对顺序。这种算法通常采用“双指针”或“快慢指针”的方法：

(1) 找到 *'e'* 合适的插入位置（秩）。

(2) 在该位置插入元素 *'e'*。

'V.put(rank, e)' 操作是替换向量中秩为 *'rank'* 的元素为 *'e'*。这不是插入操作，因为它不改变向量的大小，也不移动其他元素。因此，选项 *A* 和 *C* 是错误的。

我们需要使用 *'V.insert(rank, e)'* 操作，它会在秩 *'rank'* 处插入元素 *'e'*，并将原有该位置及之后的元素向后顺移一位。

现在关键是确定正确的插入秩 *'rank'*。在有序向量中，*'V.search(e)'* 通常实现为二分查找的一个变体。其目标是找到一个位置，以便插入 *'e'* 后向量仍然有序。一个常见的约定是 *'search(e)'* 返回秩 *'r'*，该秩是向量中不大于 *'e'* 的最大元素的秩。

- 如果 *'e'* 比向量中所有元素都小，*'search(e)'* 通常返回一个特殊值，如 *'lo - 1'* *(*对于从 *'lo'* 开始的搜索区间*)*。如果搜索整个向量 *(*从 *0* 开始*)*，则返回 *-1*。
- 如果 *'e'* 存在于向量中，*'search(e)'* 返回 *'e'* 的某个实例的秩 *'r'*。
- 如果 *'e'* 不在向量中，但 *'V[r] < e < V[r+1]'*，则 *'search(e)'* 返回 *'r'*。
- 如果 *'e'* 比向量中所有元素都大，*'search(e)'* 返回向量中最后一个元素的秩。

基于这个约定，元素 *'e'* 应该被插入到 *'search(e)'* 返回的秩 *'r'* 的下一个位置，即 *'r+1'*。

让我们分析选项 *D: 'V.insert(V.search(e) + 1, e);' Let 'r = V.search(e)'. The insertion rank will be 'r+1'.*

- 如果 *'e'* 小于向量中所有元素，*'search(e)'* 返回 *-1.* 插入秩为 *'-1 + 1 = 0'. 'V.insert(0, e)'* 得到 $\{7, 1\}$。这是正确的，*'e'* 成为新的首元素。
- 如果 *'e'* 存在于向量中，或 *'V[r] < e < V[r+1]'*，则 *'search(e)'* 返回 *'r' (*其中 *'V[r] <= e')*。插入秩为 *'r+1'. 'V.insert(r+1, e)'* 将 *'e'* 插入到 *'V[r]'* 之后，并且在任何原来大于 *'V[r]' (*且可能大于或等于 *'e')* 的元素之前。这保持了有序性。例如，若 $V = \{10, 20, 30\}$*:*
  - $e = 15$*: 'search(15)'* 返回 *0 (*秩为 *0* 的元素 *10* 是 ≤ 15 的最大元素*)*。插入秩为 $0 + 1 = 1$. *'V.insert(1, 15)'* 得到 $\{10, 15, 20, 30\}$。正确定。
  - $e = 20$*: 'search(20)'* 返回 *1 (*秩为 *1* 的元素 *20* 是 ≤ 20 的最大元素*)*。插入秩为 $1 + 1 = 2$. *'V.insert(2, 20)'* 得到 $\{10, 20, 20, 30\}$。正确定（如果允许重复）。
- 如果 *'e'* 大于向量中所有元素，*'search(e)'* 返回向量最后一个元素的秩，设为 *'hi'*。插入秩为 *'hi + 1'*。*'V.insert(hi + 1, e)'* 将 *'e'* 追加到向量末尾。这是正确的。

因此，*'V.insert(V.search(e) + 1, e)'* 是在有序向量中插入元素并保持有序的正确方法。

分析选项 *B: 'V.insert(V.search(e), e);'* 如果 $V = \{10, 20, 30\}$ 且 $e = 15$*, 'search(15)'* 返回 *0. 'V.insert(0, 15)'* 会得到 $\{15, 10, 20, 30\}$*,* 这是无序的。所以 *B* 是错误的。

21

> **对于规模为 n 的向量，低效版 uniquify() 的最坏时间复杂度为:**:
>
> ① $\Theta(n^2)$
> ② $\Theta(n \log_2 n)$
> ③ $\Theta(n)$
> ④ $\Theta(1)$

Solution 10. 正确答案为 *A*。

低效版的 *uniquify()* 通常采用双重循环：外层遍历每个元素，内层查找并删除与当前元素重复的所有元素。每次删除操作都可能导致后续元素整体前移，单次删除的时间复杂度为 $\Theta(n)$。在最坏情况下（如所有元素都相同），总共会有 $\Theta(n)$ 次删除操作，每次操作的成本也是 $\Theta(n)$，因此总的最坏时间复杂度为 $\Theta(n^2)$。

对照选项：

- *A.* $\Theta(n^2)$ ——正确
- *B.* $\Theta(n \log_2 n)$ ——错误
- *C.* $\Theta(n)$ ——错误
- *D.* $\Theta(1)$ ——错误

22

> **有序向量中的重复元素**:
> ① 与无序向量相同
> ② 大部分紧邻分布，只有极小部分散布在其它位置
> ③ 必定全部紧邻分布
> ④ 全部相间分布

Solution 11. 正确答案为 *C*。

在有序向量中，所有相等的元素必定被排在一起，全部紧邻分布。这是有序性的直接结果。无论原始数据如何，只要排序完成，所有重复元素都会连续排列，不会被其它不同元素隔开。

对照选项：

- *A.* 与无序向量相同——错误。无序向量中重复元素可能分散在任意位置。
- *B.* 大部分紧邻分布，只有极小部分散布在其它位置——错误。有序向量中不会出现这种情况。
- *C.* 必定全部紧邻分布——正确。
- *D.* 全部相间分布——错误。有序向量中不会出现重复元素被其它元素隔开的情况。

23

> **对于规模为 n 的向量，查找失败时 find() 的返回值是:**:
> ① -1
> ② 0
> ③ n
> ④ NULL

Solution 12. 正确答案为 *A*。

在大多数数据结构教材和实现（如 *C++ STL* 的 *'std::vector::find'* 或邓俊辉《数据结构》教材）中，*'find()'* 方法用于在向量中查找目标元素。如果查找成功，通常返回目标元素的秩（索引）；如果查找失败，常见的约定是返回 -1，表示未找到。

对照选项：

- *A. -1* ——正确。表示查找失败。
- *B. 0* ——错误。*0* 通常表示第一个元素的索引。
- *C. n* ——错误。*n* 通常表示向量的长度，不是合法的元素索引。
- *D. NULL* ——错误。*NULL* 不是整数类型的返回值。

因此，查找失败时 *'find()'* 的返回值是 -1。

24

> **在有序向量 V 中插入元素 e 并使之保持有序，下面代码正确的是:**:
> ① V.put(V.search(e), e);

② V.insert(V.search(e), e);
③ V.put(V.search(e) + 1, e);
④ V.insert(V.search(e) + 1, e);

Solution 13. 正确答案为 *D*。

分析如下：

- ***A. V.put(V.search(e), e);***
  错误。*put* 操作是替换指定位置的元素，不会插入新元素，也不会移动后续元素，不能保证有序性。
- ***B. V.insert(V.search(e), e);***
  错误。*search(e)* 返回的是不大于 *e* 的最大元素的秩，如果直接插入到该位置，可能会把 *e* 插入到比它小的元素前面，破坏有序性。
- ***C. V.put(V.search(e) + 1, e);***
  错误。*put* 不会插入新元素，只是替换，不能保证有序性。
- ***D. V.insert(V.search(e) + 1, e);***
  正确。*search(e)* 返回不大于 *e* 的最大元素的秩，插入到其后（即 *r+1*）可以保证 *e* 插入后仍然有序。

因此，正确的做法是 `V.insert(V.search(e) + 1, e);`。

25

在 binsearch(e, lo, hi) 版本 A 中，若 V[mi] < e，则下一步的查找范围是::

① V(mi, hi)
② V[mi, hi]
③ V(mi, hi]
④ V[lo, hi)

Solution 14. 正确答案为 *A*。

在二分查找版本 *A* 中，当发现 `V[mi] < e` 时，说明目标元素 *e*（如果存在）必然位于 *mi* 的右侧。

分析:

- 由于 `V[mi] < e`，可以确定 `V[mi]` 不是我们要找的元素。
- 在有序向量中，所有小于等于 `V[mi]` 的元素都位于 *mi* 的左侧（包括 *mi* 位置本身）。
- 因此，目标元素 *e* 只可能存在于 *mi* 的右侧，即索引大于 *mi* 的位置。
- 新的查找范围应该从 *mi+1* 开始到 *hi* 结束。

符号解释:

- `V(mi, hi)`: 表示开区间 *(mi, hi)*，即从 *mi+1* 到 *hi-1*。在某些实现中，如果原查找区间为 `[lo, hi)`（左闭右开），则新区间为 `[mi+1, hi)`，可记作 *(mi, hi)*。
- `V[mi, hi]`: 表示闭区间 `[mi, hi]`，包含 *mi* 和 *hi*。
- `V(mi, hi]`: 表示半开区间 *(mi, hi]*，不包含 *mi* 但包含 *hi*。
- `V[lo, hi)`: 表示原查找区间。

由于我们已经确定 `V[mi]` 不是目标元素，新的查找范围应该排除 *mi*，因此正确答案是 `V(mi, hi)`，即开区间。

对照选项：

- *A. V(mi, hi)* ——正确。排除了已检查的 *mi*，从 *mi+1* 开始查找。

- *B. V[mi, hi]* ——错误。仍包含已确定不匹配的 *mi*。
- *C. V(mi, hi]* ——这个表示法在不同上下文中可能正确，但选项 *A* 更标准。
- *D. V[lo, hi)* ——错误。这是原始查找范围，没有缩小。

26

> **Rank mi = (lo + hi) » 1 等效于下列哪个表达式?（lo 和 hi 非负）**:
> ① Rank mi = (lo + hi) \2
> ② Rank mi = (lo + hi) % 2
> ③ Rank mi = (lo + hi) / 2
> ④ Rank mi = (double)(lo + hi) / 2

Solution 15. 正确答案为 *C*。

分析各个选项：

- ***右移操作 '»1'***：*右移 1 位相当于将二进制数除以 2 并向下取整。对于非负整数，'x » 1' 等价于 'floor(x/2)'*。
- **选项 *A: '(lo + hi) \2'***：这里的反斜杠 *'\'* 不是标准的 *C++* 运算符。如果这是想表示整数除法，那么在某些语言中可能使用不同的符号，但在 *C++* 中不是标准语法。
- **选项 *B: '(lo + hi) % 2'***：*'%'* 是取模运算符，*'(lo + hi) % 2'* 返回的是 *'(lo + hi)'* 除以 *2* 的余数（*0* 或 *1*），而不是商。这与右移操作完全不同。
- **选项 *C: '(lo + hi) / 2'***：在 *C++* 中，当 *'lo'* 和 *'hi'* 都是整数类型（如 *'Rank'*通常是整数类型）时，*'(lo + hi) / 2'* 执行的是整数除法，结果自动向下取整。这与 *'(lo + hi) » 1'* 的行为完全一致。
- **选项 *D: '(double)(lo + hi) / 2'***：这里先将 *'(lo + hi)'* 转换为 *'double'* 类型，然后进行浮点除法。结果是一个浮点数，需要进一步转换为整数才能赋值给 *'Rank mi'*。这与右移操作不等效。

具体例子验证：

- 当 *'lo = 3, hi = 7'* 时：
  - *'(lo + hi) » 1 = 10 » 1 = 5'*
  - *'(lo + hi) / 2 = 10 / 2 = 5'*（整数除法）
  - *'(double)(lo + hi) / 2 = 10.0 / 2 = 5.0'*（浮点数）
- 当 *'lo = 2, hi = 5'* 时：
  - *'(lo + hi) » 1 = 7 » 1 = 3'*
  - *'(lo + hi) / 2 = 7 / 2 = 3'*（整数除法，向下取整）
  - *'(double)(lo + hi) / 2 = 7.0 / 2 = 3.5'*（浮点数）

结论：对于非负整数 *'lo'* 和 *'hi'*，*'(lo + hi) » 1'* 与 *'(lo + hi) / 2'*（整数除法）完全等效，都返回向下取整的商。

对照选项：

- *A. '(lo + hi) \2'* ——语法不标准
- *B. '(lo + hi) % 2'* ——错误，这是取余运算
- *C. '(lo + hi) / 2'* ——正确，整数除法等效于右移
- *D. '(double)(lo + hi) / 2'* ——错误，这是浮点除法

27

> **有序向量 V={2, 3, 5, 7, 11, 13, 17}，按二分查找算法版本 A，V.search(16, 0, 7) 需要进行多少次比较?**:
> ① 4
> ② 5
> ③ 6
> ④ 7

Solution 16. 正确答案为 *B*。

此题中的“比较”次数取决于所采用的二分查找的具体实现。要得到答案 *5*，我们可以假设“版本 *A*”是一种采用三路比较（*three-way comparison*）的实现，其逻辑如下：在每次迭代中，比较目标值 *'e'* 与中间值 *'V[mi]'*，有三种可能：*'e < V[mi]'*, *'e > V[mi]'*, 或 *'e == V[mi]'*。这在一次迭代中最多需要两次比较（例如，*'if (e < V[mi]) ... else if (e > V[mi]) ...'*）。

我们来逐步追踪这个过程：

- 向量 $V = \{2, 3, 5, 7, 11, 13, 17\}$
- 目标元素 $e = 16$
- 查找区间为 *'[0, 7)'*，即有效索引为 *'lo=0'* 到 *'hi=6'*。

***第 1 轮迭代**:*

- 当前区间: *'lo = 0, hi = 6'*
- 中点: $mi = (0 + 6)/2 = 3$
- 比较 *1:* $16 < V[3]$ *(*即 $16 < 7$*)*? 否。
- 比较 *2:* $16 > V[3]$ *(*即 $16 > 7$*)*? 是。
- 更新: 新的查找区间为 *'mi' 右*侧，*'lo = mi + 1 = 4'*。
- ***本轮比较次数: 2***

***第 2 轮迭代**:*

- 当前区间: *'lo = 4, hi = 6'*
- 中点: $mi = (4 + 6)/2 = 5$
- 比较 *1:* $16 < V[5]$ *(*即 $16 < 13$*)*? 否。
- 比较 *2:* $16 > V[5]$ *(*即 $16 > 13$*)*? 是。
- 更新: 新的查找区间为 *'mi' 右*侧，*'lo = mi + 1 = 6'*。
- ***本轮比较次数: 2***

***第 3 轮迭代**:*

- 当前区间: *'lo = 6, hi = 6'*
- 中点: $mi = (6 + 6)/2 = 6$
- 比较 *1:* $16 < V[6]$ *(*即 $16 < 17$*)*? 是。
- 更新: 新的查找区间为 *'mi' 左*侧，*'hi = mi - 1 = 5'*。
- ***本轮比较次数: 1** (*因为第一个条件就满足了，无需进行第二次比较*)*

***循环终止**:*

- 当前状态: *'lo = 6, hi = 5'*。
- *'lo > hi'*，循环条件不满足，查找结束，返回失败。

总比较次数 = $2 + 2 + 1 = 5$。

对照选项：

- *A. 4* ——错误
- *B. 5* ——正确
- *C. 6* ——错误
- *D. 7* ——错误

28

> **有序向量，V1={2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61} V2={1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18} 在 V1 中查找 43 的查找长度是 x，则在 V2 中查找 14 的查找长度为:**:
>
> ① x
> ② x+1
> ③ 2 * x
> ④ x / 2

Solution 17. 正确答案为 *A*。

核心原理：二分查找的“查找长度”(即比较次数）主要取决于两个因素:

(1) 向量的规模（长度)。

(2) 目标元素在向量中的相对位置。

它与向量中元素的具体数值无关，只要向量是有序的。

分析:

(1) **比较向量规模**:

- *V1* 的长度为 *18*。
- *V2* 的长度为 *18*。

两个向量的规模完全相同。

(2) **比较目标元素位置**:

- 在 *V1* 中，目标元素 *43* 的秩（索引）是 *13 (*因为 *V1[13] = 43)*。
- 在 *V2* 中，目标元素 *14* 的秩（索引）是 *13 (*因为 *V2[13] = 14)*。

两个目标元素在各自向量中的秩（相对位置）完全相同。

结论：由于两个向量的规模相同，并且要查找的目标元素在各自向量中的秩也相同，因此二分查找算法在这两种情况下的执行路径（即中点 *‘mi‘* 的选择序列以及 *‘lo‘* 和 *‘hi‘* 的更新过程）将是完全一样的。既然执行路径相同，那么比较的次数（查找长度）也必然相同。

因此，在 *V2* 中查找 *14* 的查找长度等于在 *V1* 中查找 *43* 的查找长度 *‘x‘*。

对照选项:

- *A. x* ——正确。
- *B. x+1* ——错误。
- *C. 2 *x* ——错误。
- *D. x / 2* ——错误。

29

> **向量 fibSearch() 算法与 binSearch() 有什么区别?**:
> ① 二者的返回值不同
> ② 前者是递归算法，后者是迭代算法
> ③ 二者选取轴点 mi 的方式不同
> ④ 前者的渐进时间复杂度低于后者，故前者效率更高

Solution 18. 正确答案为 C。

分析:

- ***fibSearch() (斐波那契查找)*** 和 ***binSearch() (二分查找)*** 都是用于有序向量的高效查找算法。它们的核心思想都是通过不断缩小查找范围来定位目标元素。

对照选项:

- ***A.* 二者的返回值不同**：错误。两种算法的目的相同，都是查找目标元素 *'e'*。因此，它们的返回值约定通常是一致的：如果找到，返回元素的秩（索引）；如果未找到，返回一个表示失败的特定值（如*-1*）。
- ***B.* 前者是递归算法，后者是迭代算法**：错误。二分查找和斐波那契查找都可以用递归或迭代两种方式实现。实现方式的选择不构成它们之间的本质区别。
- ***C.* 二者选取轴点 *mi* 的方式不同**：正确。这是两种算法最根本的区别。
  - **二分查找 *(Binary Search)***：通过将当前查找区间对半分割来选取轴点，即 $mi = lo + (hi - lo)/2$。这是一种"中分"策略。
  - **斐波那契查找 *(Fibonacci Search)***：通过斐波那契数列来分割查找区间，选取轴点的位置与黄金分割比例有关，即 $mi = lo + F_{k-1} - 1$ *(*其中 F_k 是与当前区间大小相关的斐波那契数*)*。这是一种"黄金分割"策略。

  斐波那契查找的主要优点是它只使用加法和减法来计算轴点，避免了二分查找中的除法或位移运算，这在某些计算除法代价高昂的旧式计算机上可能更快。
- ***D.* 前者的渐进时间复杂度低于后者，故前者效率更高**：错误。两种算法的渐进时间复杂度都是对数级别的。
  - 二分查找：$O(\log_2 n)$
  - 斐波那契查找：$O(\log_\phi n)$ *(*其中 $\phi \approx 1.618$ 是黄金分割比例*)*

  由于对数的底数可以相互转换（$\log_a x = \frac{\log_b x}{\log_b a}$），它们在渐进意义上属于同一个复杂度级别，即 $O(\log n)$。因此，不能说斐波那契查找的渐进时间复杂度更低。

30

> **V={1, 2, 3, 4, 5, 6, 7}，在 V 中用 Fibonacci 查找元素 1，被选取为轴点 mi 的元素依次是:**:
> ① 4, 3, 2, 1
> ② 4, 2, 1
> ③ 5, 2, 1
> ④ 5, 3, 2, 1

Solution 19. 正确答案为 D。

斐波那契查找（*Fibonacci Search*）是一种基于斐波那契数列的查找算法。其核心思想是根据斐波那契数来分割查找区间。

算法步骤追踪:

- **向量**: $V = \{1, 2, 3, 4, 5, 6, 7\}$，长度 $n = 7$。
- **目标**: $e = 1$。
- **斐波那契数**: 我们需要找到最小的斐波那契数 $F_k \geq n$。斐波那契数列为 *0, 1, 1, 2, 3, 5, 8, ...*。$F_6 = 8 \geq 7$，所以我们从 $k = 6$ 开始。
- **轴点计算**: 轴点 mi 通常由 $mi = lo + F_{k-1} - 1$ 计算得出。

第 *1* 次迭代:

- 初始区间: $[lo, hi] = [0, 6]$，$k = 6$。
- 轴点: $mi = 0 + F_{6-1} - 1 = F_5 - 1 = 5 - 1 = 4$。
- **被选取的轴点元素**: $V[4] = 5$。
- 比较: $e = 1 < V[4] = 5$。目标在左侧子区间。
- 更新: $hi = mi - 1 = 3$。k 更新为 $k - 1 = 5$。

第 *2* 次迭代:

- 当前区间: $[lo, hi] = [0, 3]$，$k = 5$。
- 轴点: $mi = 0 + F_{5-1} - 1 = F_4 - 1 = 3 - 1 = 2$。
- **被选取的轴点元素**: $V[2] = 3$。
- 比较: $e = 1 < V[2] = 3$。目标在左侧子区间。
- 更新: $hi = mi - 1 = 1$。k 更新为 $k - 1 = 4$。

第 *3* 次迭代:

- 当前区间: $[lo, hi] = [0, 1]$，$k = 4$。
- 轴点: $mi = 0 + F_{4-1} - 1 = F_3 - 1 = 2 - 1 = 1$。
- **被选取的轴点元素**: $V[1] = 2$。
- 比较: $e = 1 < V[1] = 2$。目标在左侧子区间。
- 更新: $hi = mi - 1 = 0$。k 更新为 $k - 1 = 3$。

第 *4* 次迭代:

- 当前区间: $[lo, hi] = [0, 0]$，$k = 3$。
- 轴点: $mi = 0 + F_{3-1} - 1 = F_2 - 1 = 1 - 1 = 0$。
- **被选取的轴点元素**: $V[0] = 1$。
- 比较: $e = 1 == V[0] = 1$。找到目标。

综上，被选取的轴点元素依次是 *5, 3, 2, 1*。

对照选项:

- *A. 4, 3, 2, 1* - 错误
- *B. 4, 2, 1* - 错误
- *C. 5, 2, 1* - 错误
- *D. 5, 3, 2, 1* - 正确

31

> **如果（有序）向量中元素的分布满足独立均匀分布（排序前），插值查找的平均时间复杂度为:**:
>
> ① O(n)
> ② O(logn)
> ③ O(loglogn)

④ O(1)

Solution 20. 正确答案为 C。

分析:

- **插值查找 *(Interpolation Search)*** 是一种对二分查找的改进算法，它利用了数据分布的信息来更智能地猜测目标元素的位置。
- **二分查找** 总是检查区间的中间点，即 $mi = lo + (hi - lo)/2$。
- **插值查找** 根据目标值 *'e'* 在当前区间端点值 *'V[lo]'* 和 *'V[hi]'* 之间的相对位置来预测其索引，公式为 $mi = lo + \lfloor \frac{(e-V[lo])\cdot(hi-lo)}{V[hi]-V[lo]} \rfloor$。

关键条件:

- 算法的效率高度依赖于数据的分布。
- 题目中明确指出，元素的分布满足 **独立均匀分布**。这是插值查找表现最佳的理想情况。

复杂度分析:

- 在数据均匀分布的假设下，插值查找的查找范围收缩得非常快。平均而言，每次迭代后，查找范围的大小会缩小到其原大小的平方根（即从 n 到 $\sqrt{n}$）。
- 这种快速的收缩导致其平均时间复杂度为 $O(\log(\log n))$。
- 相比之下，二分查找的平均和最坏时间复杂度都是 $O(\log n)$。
- 需要注意的是，插值查找的最坏情况时间复杂度是 $O(n)$，这发生在数据分布极不均匀的情况下（例如，指数增长的序列）。但本题问的是平均情况。

对照选项:

- *A. O(n)* - 这是插值查找的最坏情况复杂度。
- *B. O(logn)* - 这是二分查找的复杂度。
- *C. O(loglogn)* - 正确。这是插值查找在均匀分布下的平均复杂度。
- *D. O(1)* - 错误。

32

在向量 V={2, 3, 5, 7, 11, 13, 17, 19, 23} 中用插值查找搜索元素 7，猜测的轴点 mi 依次是::

① 1, 2, 3
② 2, 3
③ 1, 3
④ 3

Solution 21. 正确答案为 A。

向量 $V = \{2, 3, 5, 7, 11, 13, 17, 19, 23\}$。目标元素 $e = 7$。数组大小 $n = 9$。

插值查找的轴点计算公式为: $mi = lo + \lfloor \frac{(e-V[lo])\cdot(hi-lo)}{V[hi]-V[lo]} \rfloor$。

迭代 *1*:

- *初始:* $lo = 0, hi = 8$。
- $V[lo] = V[0] = 2$。
- $V[hi] = V[8] = 23$。
- $mi = 0 + \lfloor \frac{(7-2)\cdot(8-0)}{23-2} \rfloor = 0 + \lfloor \frac{5\cdot 8}{21} \rfloor = 0 + \lfloor \frac{40}{21} \rfloor = 0 + \lfloor 1.9047... \rfloor = 0 + 1 = 1$。
- 轴点元素 $V[mi] = V[1] = 3$。

- 比较 $e = 7$ 与 $V[1] = 3$。因为 $7 > 3$，目标在右侧。
- 更新 $lo = mi + 1 = 1 + 1 = 2$。hi 保持为 *8*。
- 本轮轴点: $mi = 1$ *(元素值为 3)*。

迭代 *2*:

- 当前: $lo = 2, hi = 8$。
- $V[lo] = V[2] = 5$。
- $V[hi] = V[8] = 23$。
- $mi = 2 + \lfloor \frac{(7-5)\cdot(8-2)}{23-5} \rfloor = 2 + \lfloor \frac{2\cdot 6}{18} \rfloor = 2 + \lfloor \frac{12}{18} \rfloor = 2 + \lfloor 0.666... \rfloor = 2 + 0 = 2$。
- 轴点元素 $V[mi] = V[2] = 5$。
- 比较 $e = 7$ 与 $V[2] = 5$。因为 $7 > 5$，目标在右侧。
- 更新 $lo = mi + 1 = 2 + 1 = 3$。hi 保持为 *8*。
- 本轮轴点: $mi = 2$ *(元素值为 5)*。

迭代 *3*:

- 当前: $lo = 3, hi = 8$。
- $V[lo] = V[3] = 7$。
- $V[hi] = V[8] = 23$。
- $mi = 3 + \lfloor \frac{(7-7)\cdot(8-3)}{23-7} \rfloor = 3 + \lfloor \frac{0\cdot 5}{16} \rfloor = 3 + \lfloor 0 \rfloor = 3 + 0 = 3$。
- 轴点元素 $V[mi] = V[3] = 7$。
- 比较 $e = 7$ 与 $V[3] = 7$。因为 $7 == 7$，找到元素。
- 本轮轴点: $mi = 3$ *(元素值为 7)*。

猜测的轴点 *mi (索引)* 依次是 *1, 2, 3*。

33

> **对于规模为 n 的向量，二分查找版本 A 和 B 的最优时间复杂度分别为:**:
>
> ① $\Theta(n^2), \Theta(n)$
> ② $\Theta(n \log_2 n), \Theta(n)$
> ③ $\Theta(\log_2 n), \Theta(\log_2 n)$
> ④ $O(1), \Theta(\log_2 n)$

Solution 22. 正确答案为 D。

二分查找版本 *A* (标准版本): 标准二分查找算法在每次迭代时比较目标元素与当前区间的中间元素。

- **最优情况**: 当目标元素恰好是数组的第一个中间元素时，仅需一次比较即可找到。
- **最优时间复杂度**: $\Theta(1)$。

二分查找版本 *B* (特定变体): 二分查找存在多种变体。其中一些变体（例如，某些教材中为了确保找到不大于/不小于目标值的最大/最小元素，或者总是将区间缩小到特定大小的变体）可能设计为循环总是执行对数次数的操作，即使目标元素可能在早期被“匹配”。

- 例如，邓俊辉《数据结构（*C++* 语言版）》中介绍的二分查找版本 *B*（如图 *3.3(b)*），其循环部分（用以确定秩）无论目标元素为何或是否存在，都会执行 $\Theta(\log_2 n)$ 次比较。
- **最优情况 (对于这类变体的操作次数)**: 对于这类固定执行对数次迭代的变体，其操作次数（包括最优情况）是 $\Theta(\log_2 n)$。

分析选项:

- *A.* $\Theta(n^2), \Theta(n)$*:* 不正确。
- *B.* $\Theta(n \log_2 n), \Theta(n)$*:* 不正确。
- *C.* $\Theta(\log_2 n), \Theta(\log_2 n)$*:* 对于版本 *A* 的最优情况不正确。
- *D.* $O(1), \Theta(\log_2 n)$*:*
  - 版本 *A* 的最优时间复杂度为 $\Theta(1)$。$O(1)$ 是 $\Theta(1)$ 的一个有效上界表示。
  - 版本 *B*（特指上述类型的变体）的最优（也是其固有）时间复杂度为 $\Theta(\log_2 n)$。

  此选项与分析相符。

因此，对于规模为 *n* 的向量，二分查找版本 *A* 的最优时间复杂度为 $O(1)$ *(*更精确地是 $\Theta(1)$*)*，版本 *B*（特定变体）的最优时间复杂度为 $\Theta(\log_2 n)$。

34

> **对于 search() 接口，我们约定当向量中存在多个目标元素时返回其中:**:
>
> ① 秩最大者
> ② 秩最小者
> ③ 秩中间者
> ④ 随机返回其中一个即可

Solution 23. 正确答案为 *A*。

当有序向量中存在多个与目标元素 *'e'* 相等的元素时，*'search()'* 接口返回哪一个元素的秩（索引）取决于具体的约定或算法实现。

- ***A.* 秩最大者***:* 返回目标元素中秩（索引）最大的那一个。某些查找算法的实现（特别是一些二分查找的变体，如邓俊辉《数据结构》中旨在查找"不大于 *e* 的最大元素"的版本）在找到目标元素 *'e'* 时，自然会定位到多个相同元素中的最后一个。如果约定 *'search()'* 接口遵循此类行为，则会返回秩最大者。
- ***B.* 秩最小者***:* 返回目标元素中秩（索引）最小的那一个。这是许多标准库函数（如 *C++ STL* 中的 *'std::find'* 或 *Python* 的 *'list.index()'*）的常见行为。对于线性扫描，这通常是最先被找到的元素。
- ***C.* 秩中间者***:* 返回秩居中的目标元素。这种约定不常见，且在有偶数个目标元素时定义不唯一，实现也相对复杂。
- ***D.* 随机返回其中一个即可***:* 虽然某些简单实现可能表现出这种行为（例如，标准二分查找在命中后立即返回，具体返回哪一个取决于中间点计算和元素分布），但这通常不被视为一个正式的、可靠的接口约定。

考虑到这些问题的上下文可能源于特定的教材（如邓俊辉的《数据结构》），其推荐的二分查找版本（例如版本 *B* 或 *C*）在设计上倾向于找到满足特定条件的边界元素。例如，查找"不大于 *e* 的元素中秩最大者"，如果目标元素 *e* 存在，该算法会返回 *e* 的最后一个出现位置（即秩最大者）。因此，如果"我们约定"指的是该教材或课程中的规范，那么选项 *A* 是合理的。

如果一个 *'search()'* 接口需要一个明确且唯一的返回值，约定返回秩最大者或秩最小者都是确定性的选择。在此特定选择题的语境下，选择 *A* 通常意味着所参考的查找算法实现（尤其是二分查找的某种特定版本）在元素存在时会收敛到最后一个匹配项。

35

> **对于二分查找版本 C，当 e<V[mi] 不成立时下一步的查找范围是:**:
>
> ① V[lo, mi)
> ② V[mi, hi)
> ③ V[mi, hi]
> ④ V(mi, hi)

Solution 24. 正确答案为 D。

二分查找版本 C 通常指的是一种旨在查找有序向量 V 中第一个不小于目标值 e 的元素的位置（或者说，满足 $V[k] \geq e$ 的最小索引 k）。其搜索区间通常维护为左闭右开的形式，即 $[lo, hi)$。

算法步骤如下：

(1) 当 $lo < hi$ 时，循环继续。

(2) 计算中间点 $mi = lo + \lfloor (hi - lo)/2 \rfloor$。

(3) 比较 e 与 $V[mi]$:

- 如果 $e < V[mi]$: 目标元素 e (或者第一个不小于 e 的元素) 必然在 mi 的左侧（可能包括 mi 自身，如果 $V[mi]$ 是第一个不小于 e 的元素）。因此，新的搜索范围是 $[lo, mi)$。这通过更新 $hi = mi$ 实现。
- 如果 $e \geq V[mi]$ (即题目中的“$e < V[mi]$ 不成立”): 这意味着 $V[mi]$ 小于或等于 e。由于我们要找的是第一个不小于 e 的元素，所以 $V[mi]$ 以及 $V[mi]$ 之前的所有元素都不可能是最终答案（除非 $V[mi]$ 恰好是 e 且它是满足条件的第一个，但为了保证找到的是“第一个”，我们需要继续向右搜索）。因此，新的搜索范围必须从 mi 的右侧开始，即 $[mi + 1, hi)$。这通过更新 $lo = mi + 1$ 实现。

当循环结束时 ($lo = hi$)，lo (或 hi) 就是第一个不小于 e 的元素的位置。

题目条件是“$e < V[mi]$ 不成立”，即 $e \geq V[mi]$。根据上述逻辑，此时应更新 $lo = mi + 1$。因此，下一步的查找范围是 $[mi + 1, hi)$。

对照选项：

- *A.* $V[lo, mi)$: 表示区间 $[lo, mi)$。这是当 $e < V[mi]$ 时的情况。
- *B.* $V[mi, hi)$: 表示区间 $[mi, hi)$。这是当 $lo = mi$ 时的情况，例如在某些版本的二分查找中，如果 $e \geq V[mi]$ 且要找最后一个不大于 e 的元素。
- *C.* $V[mi, hi]$: 表示区间 $[mi, hi]$ (全闭区间)。
- *D.* $V(mi, hi)$: 此符号通常表示开区间 (mi, hi)。在数组索引的上下文中，如果 hi 是原区间的开上界，则 (mi, hi) 对应于索引从 $mi + 1$ 到 $hi - 1$ 的元素，即区间 $[mi + 1, hi)$。这与我们推导出的新搜索范围 $[mi + 1, hi)$ 相符。

因此，当 $e < V[mi]$ 不成立时，下一步的查找范围是 $V(mi, hi)$，即 $[mi + 1, hi)$。

36

> **向量起泡排序中，V={7, 2, 3, 11, 17, 5, 19, 13}，对 V 进行两次扫描交换后 V[6] =**:
>
> ① 11
> ② 13

> ③ 17
> ④ 19

Solution 25. 正确答案为 C。

初始向量 $V = \{7, 2, 3, 11, 17, 5, 19, 13\}$。向量长度 $n = 8$。索引从 *0* 到 *7*。

第一次扫描 (Pass 1): 目的是将最大的元素冒泡到最右端 $V[7]$。

- 比较 $V[0] = 7$ 和 $V[1] = 2$。$7 > 2$，交换。$V = \{2, 7, 3, 11, 17, 5, 19, 13\}$
- 比较 $V[1] = 7$ 和 $V[2] = 3$。$7 > 3$，交换。$V = \{2, 3, 7, 11, 17, 5, 19, 13\}$
- 比较 $V[2] = 7$ 和 $V[3] = 11$。$7 < 11$，不交换。$V = \{2, 3, 7, 11, 17, 5, 19, 13\}$
- 比较 $V[3] = 11$ 和 $V[4] = 17$。$11 < 17$，不交换。$V = \{2, 3, 7, 11, 17, 5, 19, 13\}$
- 比较 $V[4] = 17$ 和 $V[5] = 5$。$17 > 5$，交换。$V = \{2, 3, 7, 11, 5, 17, 19, 13\}$
- 比较 $V[5] = 17$ 和 $V[6] = 19$。$17 < 19$，不交换。$V = \{2, 3, 7, 11, 5, 17, 19, 13\}$
- 比较 $V[6] = 19$ 和 $V[7] = 13$。$19 > 13$，交换。$V = \{2, 3, 7, 11, 5, 17, 13, 19\}$

第一次扫描后，$V = \{2, 3, 7, 11, 5, 17, 13, 19\}$。此时 $V[6] = 13$。

第二次扫描 (Pass 2): 目的是将次大的元素冒泡到 $V[6]$ *(因为 $V[7]$ 已经是最大值，不再参与比较)*。扫描范围是 $V[0 \ldots 6]$。

- 比较 $V[0] = 2$ 和 $V[1] = 3$。$2 < 3$，不交换。$V = \{2, 3, 7, 11, 5, 17, 13, 19\}$
- 比较 $V[1] = 3$ 和 $V[2] = 7$。$3 < 7$，不交换。$V = \{2, 3, 7, 11, 5, 17, 13, 19\}$
- 比较 $V[2] = 7$ 和 $V[3] = 11$。$7 < 11$，不交换。$V = \{2, 3, 7, 11, 5, 17, 13, 19\}$
- 比较 $V[3] = 11$ 和 $V[4] = 5$。$11 > 5$，交换。$V = \{2, 3, 7, 5, 11, 17, 13, 19\}$
- 比较 $V[4] = 11$ 和 $V[5] = 17$。$11 < 17$，不交换。$V = \{2, 3, 7, 5, 11, 17, 13, 19\}$
- 比较 $V[5] = 17$ 和 $V[6] = 13$。$17 > 13$，交换。$V = \{2, 3, 7, 5, 11, 13, 17, 19\}$

第二次扫描后，$V = \{2, 3, 7, 5, 11, 13, 17, 19\}$。此时 $V[6] = 17$。

对照选项:

- *A. 11*
- *B. 13*
- *C. 17*
- *D. 19*

因此，对 *V* 进行两次扫描交换后 *V[6] = 17*。

37

> **经改进的起泡排序在什么情况下会提前结束?**:
> ① 完成全部 n-1 趟扫描交换
> ② 完成的扫描交换趟数 = 实际发生元素交换的扫描交换趟数 + 1
> ③ 完成的扫描交换趟数 = 实际发生元素交换的扫描交换趟数
> ④ 完成 (n - 1) / 2 趟扫描交换

Solution 26. 正确答案为 *B*。

标准的起泡排序（*Bubble Sort*）无论输入数据如何，都会执行 $n - 1$ 趟扫描交换。

经改进的起泡排序引入了一个标志位（*flag*），用于记录在某一趟扫描中是否发生了元素交换。

- 在每一趟扫描开始前，将标志位设置为假（表示尚未发生交换）。

- 如果在扫描过程中发生了任何元素交换，则将标志位设置为真。
- 在一趟扫描结束后，如果标志位仍然为假，说明在这一整趟扫描中没有发生任何元素交换。这意味着所有元素都已处于其最终的有序位置，数组已经排好序。此时，算法可以提前终止，无需再进行后续的扫描。

分析选项：

- ***A. 完成全部 n-1 趟扫描交换**: 这是未改进的起泡排序的行为，或者改进版在最坏情况（例如，完全逆序的数组）下的行为。它不是提前结束的条件。*
- ***B. 完成的扫描交换趟数 = 实际发生元素交换的扫描交换趟数 + 1**: 这准确地描述了提前结束的条件。算法会执行若干趟扫描，在这些趟中至少发生了一次交换（这些是"实际发生元素交换的扫描交换趟数"）。然后，它会再执行一趟扫描，在这一趟中没有发生任何交换。这一趟"无交换"的扫描确认了数组已排序，于是算法终止。因此，总的扫描趟数是发生交换的趟数加 1（这最后一次无交换的确认趟）。*
- ***C. 完成的扫描交换趟数 = 实际发生元素交换的扫描交换趟数**: 这不正确。如果算法在最后一趟发生交换后立即停止，它无法确认数组是否真的完全排序。需要额外一趟无交换的扫描来确认。*
- ***D. 完成 (n - 1) / 2 趟扫描交换**: 这不是一个通用的提前结束条件。提前结束取决于数据的有序程度，而不是固定的趟数比例。*

因此，改进的起泡排序在"某一趟扫描结束后，发现该趟没有进行任何元素交换"时提前结束。这对应于选项 *B* 的描述。

38

> **试用以下算法对 V={19, 17, 23} 排序：**
>
> **(1) 先按个位排序**
>
> **(2) 在上一步基础上，再按十位排序**
>
> **这个算法是否正确?**：
>
> ① 一定正确
>
> ② 一定不正确
>
> ③ 若第 2 步用的排序算法是稳定的，则正确
>
> ④ 若第 1 步用的排序算法是稳定的，则正确

Solution 27. 正确答案为 *C*。

该算法描述的是一种两遍的基数排序（*LSD Radix Sort*，最低位优先）。初始向量 $V = \{19, 17, 23\}$。

第 1 步：按个位排序

- *19 (个位是 9)*
- *17 (个位是 7)*
- *23 (个位是 3)*

按个位从小到大排序后，向量变为 $V_1 = \{23, 17, 19\}$。*(对于此特定输入，由于个位数都不同，第一步排序的稳定性不影响此中间结果。)*

第 2 步：在上一步基础上 ($V_1 = \{23, 17, 19\}$)，再按十位排序

- *23 (十位是 2)*
- *17 (十位是 1)*
- *19 (十位是 1)*

现在需要按十位对 $V_1 = \{23, 17, 19\}$ 进行排序。

- 元素 *17* 和 *19* 的十位相同（都是 *1*）。
- 元素 *23* 的十位是 *2*。

为了使最终排序正确，当比较十位时，如果十位数相同（如 *17* 和 *19*），它们之间原有的、基于个位数的排序结果必须被保留。在 V_1 中，*17 (*个位 *7) 在 19 (*个位 *9)* 之前。

- **如果第 *2* 步的排序算法是稳定的**：当处理十位为 *1* 的元素 *(17, 19)* 时，稳定的排序算法会保持它们在 V_1 中的相对顺序，即 *17* 仍然在 *19* 之前。因此，排序结果将是 $\{17, 19, 23\}$。这是正确的排序。
- **如果第 *2* 步的排序算法是不稳定的**：当处理十位为 *1* 的元素 *(17, 19)* 时，不稳定的排序算法可能会改变它们的相对顺序，可能将它们排成 $\{19, 17\}$。那么最终结果可能是 $\{19, 17, 23\}$。这是不正确的排序。

结论：基数排序（尤其是 *LSD*，如此处描述的）要求每一轮对单个“数字位”的排序都必须是稳定的。这是因为后续的排序（更高位）依赖于前序排序（更低位）已经建立的相对顺序，对于当前位相同的元素，这个相对顺序不能被打乱。在这个两步算法中，第二步（按十位排序）是关键。如果第二步的排序是稳定的，它会正确地将元素按十位排序，并且对于十位相同的元素（如 *17* 和 *19*），会保留它们在第一步按个位排序后形成的顺序（*17* 先于 *19*）。

- ***A.* 一定正确**：错误，取决于排序算法的稳定性。
- ***B.* 一定不正确**：错误，如果使用稳定排序则可能正确。
- ***C.* 若第 *2* 步用的排序算法是稳定的，则正确**：正确。这是 *LSD* 基数排序正确性的核心要求。
- ***D.* 若第 *1* 步用的排序算法是稳定的，则正确**：不充分。即使第 *1* 步稳定，如果第 *2* 步不稳定，算法仍可能失败。第 *1* 步的稳定性主要确保如果原始数据中有值完全相同的项，它们的原始相对顺序在第一遍后被保持（如果这个算法的整体目标是稳定排序的话）。但对于数值本身的正确排序，第 *2* 步的稳定性更为关键。

因此，该算法的正确性依赖于第 *2* 步（对更高位进行排序的那一步）所使用的排序算法是稳定的。

39

> **对 {2, 5, 7} 和 {3, 11, 13} 进行二路归并，执行的元素比较依次是:**:
> ① 2 与 3、5 与 3、5 与 11、7 与 11
> ② 2 与 5、5 与 3、11 与 13、5 与 7
> ③ 2 与 3、2 与 11、7 与 11、13 与 7
> ④ 2 与 3、5 与 11、7 与 13

Solution 28. 正确答案为 *A*。

二路归并（*2-way merge*）是将两个已排序的序列合并成一个有序序列的过程。设两个待归并的有序向量为 $V_1 = \{2, 5, 7\}$ 和 $V_2 = \{3, 11, 13\}$。我们使用两个指针，分别指向 V_1 和 V_2 的当前待比较元素。

***(1)* 初始状态**:

- $V_1 = \{2, 5, 7\}$ *(*指针 p_1 指向 *2)*
- $V_2 = \{3, 11, 13\}$ *(*指针 p_2 指向 *3)*
- 结果序列 $R = \{\}$

(2)* 比较 *1:

- 比较 $V_1[p_1] = 2$ 与 $V_2[p_2] = 3$。
- 因为 $2 < 3$，将 *2* 加入 R。

- $R = \{2\}$。p_1 后移，指向 *5*。
- 本轮比较: ***2* 与 *3***。

***(3)* 比较 *2*:**

- 比较 $V_1[p_1] = 5$ 与 $V_2[p_2] = 3$。
- 因为 $5 > 3$，将 *3* 加入 R。
- $R = \{2, 3\}$。p_2 后移，指向 *11*。
- 本轮比较: ***5* 与 *3***。

***(4)* 比较 *3*:**

- 比较 $V_1[p_1] = 5$ 与 $V_2[p_2] = 11$。
- 因为 $5 < 11$，将 *5* 加入 R。
- $R = \{2, 3, 5\}$。p_1 后移，指向 *7*。
- 本轮比较: ***5* 与 *11***。

***(5)* 比较 *4*:**

- 比较 $V_1[p_1] = 7$ 与 $V_2[p_2] = 11$。
- 因为 $7 < 11$，将 *7* 加入 R。
- $R = \{2, 3, 5, 7\}$。p_1 后移，超出 V_1 范围。
- 本轮比较: ***7* 与 *11***。

***(6)* 结束:**

- V_1 已全部处理完毕。将 V_2 中剩余的元素 *(11, 13)* 直接加入 R。
- $R = \{2, 3, 5, 7, 11, 13\}$。
- 此步骤不涉及新的比较。

执行的元素比较依次是: *2* 与 *3*、*5* 与 *3*、*5* 与 *11*、*7* 与 *11*。

对照选项:

- *A. 2* 与 *3*、*5* 与 *3*、*5* 与 *11*、*7* 与 *11* - 正确。
- *B. 2* 与 *5*、*5* 与 *3*、*11* 与 *13*、*5* 与 *7* - 错误。
- *C. 2* 与 *3*、*2* 与 *11*、*7* 与 *11*、*13* 与 *7* - 错误。
- *D. 2* 与 *3*、*5* 与 *11*、*7* 与 *13* - 错误。

40

> **对于规模为 n 的向量，归并排序的最优、最坏时间复杂度分别为:**:
>
> ① $\Theta(n), \Theta(n \log_2 n)$
> ② $\Theta(n \log_2 n), \Theta(n \log_2 n)$
> ③ $\Theta(n \log_2 n), \Theta(n^2)$
> ④ $\Theta(n), \Theta(n^2)$

Solution 29. 正确答案为 *B*。

归并排序（*Merge Sort*）是一种分治算法。其基本思想如下:

***(1)* 分解（*Divide*）**: 将包含 n 个元素的序列递归地分成两个各含 $n/2$ 个元素的子序列。

***(2)* 解决（*Conquer*）**: 递归地排序两个子序列。如果子序列的大小为 *1*，则它们自然有序。

***(3)* 合并（*Combine*）**: 合并两个已排序的子序列以产生排序好的答案。这一步是核心，需要 $O(n)$ 时间。

时间复杂度分析:

- 归并排序的递归关系可以表示为 $T(n) = 2T(n/2) + \Theta(n)$，其中 $\Theta(n)$ 是合并步骤的成本。
- 根据主定理（*Master Theorem*），这个递归式的解是 $T(n) = \Theta(n \log_2 n)$。
- 重要的是，归并排序的行为不依赖于输入数据的初始顺序。无论数据是已经排序、逆序排序还是随机排列，分解和合并步骤的操作次数都保持一致。
  - **分解**：总是将序列对半划分，这需要 $\log_2 n$ 层递归。
  - **合并**：在每一层递归中，合并所有子序列的总工作量是 $\Theta(n)$。
- 因此，归并排序在最优情况、平均情况和最坏情况下的时间复杂度都是 $\Theta(n \log_2 n)$。

对照选项：

- *A.* $\Theta(n), \Theta(n \log_2 n)$*:* 错误。最优情况也是 $\Theta(n \log_2 n)$。
- *B.* $\Theta(n \log_2 n), \Theta(n \log_2 n)$*:* 正确。
- *C.* $\Theta(n \log_2 n), \Theta(n^2)$*:* 错误。最坏情况不是 $\Theta(n^2)$。
- *D.* $\Theta(n), \Theta(n^2)$*:* 错误。

因此，对于规模为 *n* 的向量，归并排序的最优和最坏时间复杂度均为 $\Theta(n \log_2 n)$。