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 第三章列表

41

> **下列关于列表的秩的说法中不正确的是**:
>
> ① 列表节点的秩与物理地址有明确的对应关系
>
> ② 列表的秩具有很高的维护成本
>
> ③ 在列表中循秩访问的成本较高
>
> ④ 在列表中循位置访问会比循秩访问更快

Solution 1. 正确答案为 A。

"列表"可以指多种数据结构，主要包括基于数组的列表（如向量）和基于指针的列表（如链表）。"秩"是指元素在列表中的逻辑位置或序号。

- ***A.* 列表节点的秩与物理地址有明确的对应关系**:
  - 对于**基于数组的列表**，元素的秩（即索引）与其物理地址之间存在明确的线性关系：'地址 *(*秩 *r)* = 基地址 + *r* * 元素大小 '。所以对于数组列表，此说法正确。
  - 对于**链表**，节点在内存中可以任意分布，节点的秩与其物理地址之间没有直接的、简单的算术对应关系。要访问特定秩的节点，必须从头节点开始遍历。所以对于链表，此说法**不正确**。
  - 由于该说法并非对所有类型的列表都成立（特别是对链表不成立），因此作为关于"列表"的一般性陈述，它是**不正确的**。
- ***B.* 列表的秩具有很高的维护成本**:
  - 秩本身是逻辑概念。如果指的是在插入或删除操作后，需要更新显式存储在每个节点中的秩信息，那么对于链表和数组列表，这都可能导致 $\Theta(n)$ 的成本。
  - 对于数组列表，在特定秩插入或删除元素通常需要移动 $\Theta(n)$ 个元素，这也是一种高成本。
  - 因此，此说法可以被认为是**正确的**，因为维护有序性和元素位置（从而影响秩）的操作可能成本很高。
- ***C.* 在列表中循秩访问的成本较高**:
  - 对于**链表**，访问秩为 r 的元素需要从头遍历 r 个节点，时间复杂度为 $\Theta(r)$（最坏为 $\Theta(n)$），成本较高。
  - 对于**基于数组的列表**，通过秩（索引）访问元素的时间复杂度为 $\Theta(1)$，成本较低。
  - 由于此说法对链表成立（成本较高），而题目通常会考虑各种列表实现，链表的这种特性使得该说法在某些情况下是**正确的**（即存在成本较高的情况）。如果题目问的是普遍性，那么它对数组列表不正确。但通常这类表述会关注到链表的特性。
- ***D.* 在列表中循位置访问会比循秩访问更快**:
  - "循位置访问"通常指已知节点的直接引用（如指针或迭代器）。通过指针/迭代器访问节点数据的成本是 $\Theta(1)$。
  - 对于**链表**，循秩访问是 $\Theta(n)$，循位置访问是 $\Theta(1)$。因此循位置访问更快。
  - 对于**基于数组的列表**，循秩（索引）访问是 $\Theta(1)$。"循位置访问"如果也指通过索引访问，则两者速度相同。如果"位置"特指一个已经获取到的指针或引用，那么也是 $\Theta(1)$。
  - 考虑到链表的情况，其中差异显著，此说法通常被认为是**正确的**。

结论：语句 A 声称秩与物理地址有明确对应关系，这仅对数组型列表成立，对链式列表则完全不成立。因此，作为对"列表"的通用描述，语句 A 是不正确的。其他选项描述了列表（尤其是链表或在某些操作下）可

能具有的特性或与其他访问方式的比较，这些在特定场景下是成立的。因此，最不正确的说法是 A。

42

> **下列关于我们定义的接口的描述中，哪一条是错误的?**:
> ① 对于列表中的节点，我们可以通过调用 pred() 和 succ() 接口分别取其前驱和后继。
> ② 对于列表接口中的 find(e) 与 search(e)，其中一个重要区别在于 find 普适于所有列表，而 search 适用于有序列表。
> ③ 如果一个列表的 visible list 部分长度为 n，则头、首、末、尾节点的秩分别为 -1, 0, n, n + 1
> ④ 在构造列表时，我们需要首先构造哨兵节点。

Solution 2. 正确答案为 C。

让我们逐条分析:

- ***A.* 对于列表中的节点，我们可以通过调用 *pred()* 和 *succ()* 接口分别取其前驱和后继。**
  - 这是列表节点(尤其是链表节点)非常标准的接口。*'pred()'* 返回前驱节点的引用/指针，*'succ()'* 返回后继节点的引用/指针。
  - 此说法是**正确的**。
- ***B.* 对于列表接口中的 *find(e)* 与 *search(e)*，其中一个重要区别在于 *find* 普适于所有列表，而 *search* 适用于有序列表。**
  - *'find(e)'* 通常指在列表中查找元素 *'e'*，不依赖列表的有序性，一般从头到尾顺序查找。因此它普适于所有列表（有序或无序）。
  - *'search(e)'* 通常指更高效的查找算法，如二分查找。这类算法要求列表是有序的。
  - 此说法是**正确的**。
- ***C.* 如果一个列表的 *visible list* 部分长度为 *n*，则头、首、末、尾节点的秩分别为 *-1, 0, n, n + 1***
  - "*Visible list* 部分长度为 *n*"指的是列表中有 n 个实际的数据元素。
  - 秩通常是 *0-indexed*。所以 n 个元素的秩范围是 $0, 1, \ldots, n-1$.
  - "头"*(header sentinel)*: 通常约定其秩为 *-1*。
  - "首"*(first data node)*: 其秩为 *0*。
  - "末"*(last data node)*: 其秩为 $n-1$。
  - "尾"*(trailer sentinel)*: 逻辑上位于最后一个数据元素之后，其秩应为 n。
  - 语句中给出的秩:
    - 头 *(header): -1* (符合约定)
    - 首 *(first data node): 0* (符合约定)
    - 末 *(last data node):* n (不符合约定，应为 $n-1$)
    - 尾 *(trailer):* $n+1$ (不符合约定，应为 n)
  - 因此，此说法是**错误的**。
- ***D.* 在构造列表时，我们需要首先构造哨兵节点。**
  - 对于使用哨兵节点（如头哨兵 *header* 和尾哨兵 *trailer*）的列表实现，在列表初始化时创建这些哨兵节点是标准做法。哨兵节点的存在可以简化插入、删除等操作的边界条件处理。
  - 此说法是**正确的**（对于采用哨兵节点设计的列表而言）。

综上所述，说法 C 是错误的，因为它与常见约定不符：在一个有 n 个元素的列表中，最后一个数据节点的秩应为 $n-1$，而尾哨兵的秩应为 n。

43

若将 insertAsPred() 改为以下函数,

```
template <typename T>
ListNodeosi(T) ListNode<T>::insertAsPred(T const & e){
    ListNodeosi(T)  x=new ListNode(e,pred,this);
    pred = x;
    pred->succ = x;
    return x;
}
```

其结果是:

① 能正常插入节点
② 不能插入节点,原列表仍然保持不变
③ 不能插入节点,原列表的结构被破坏
④ 能否插入节点与当时列表的结构有关

Solution 3. 正确答案为 C。

让我们分析这段代码的执行过程。假设 '*ListNodeosi(T)*' 是 '*ListNode<T>**' 的别名,代表指向列表节点的指针。'*this*' 是调用 '*insertAsPred*' 方法的节点,我们称之为 '*N*'。`this->pred` 是 '*N*' 的前驱节点,我们称之为 `P_orig` (在函数开始时)。

(1) '*ListNodeosi(T) x = new ListNode(e, pred, this);*'

- 创建一个新节点 '*x*'。
- '*x->data*' 被设置为 '*e*'。
- `x->pred` 被设置为 `this->pred` (即 `P_orig`)。
- '*x->succ*' 被设置为 '*this*' (即 '*N*')。
- 至此,新节点 '*x*' 初始化时,其前驱指向 '*N*' 的原前驱 `P_orig`,其后继指向 '*N*'。
- 此时的链接关系(部分): `P_orig` $\leftarrow x \rightarrow N$。节点 `P_orig` 和 N 之间的原始链接仍然存在。

(2) '*pred = x;*'

- 这一行将 '*this->pred*' (即 '*N->pred*') 修改为指向新节点 '*x*'。
- 现在 '*N*' 的前驱是 '*x*'。
- 此时的链接关系(部分): `P_orig` $x \leftarrow N$。'*x->pred*' 仍然是 `P_orig`,'*x->succ*' 仍然是 '*N*'。

(3) '*pred->succ = x;*'

- 这里的 '*pred*' 是 '*this->pred*',它在上一行已经被修改为指向 '*x*'。
- 所以,这一行实际上是执行 '*x->succ = x;*'。
- 这使得新节点 '*x*' 的后继指针指向它自身,形成一个自循环。
- 原本在步骤 *1* 中设置的 '*x->succ = N*' 被覆盖了。

最终的链接状态分析: 假设原始列表片段为 $\ldots \leftrightarrow P_{orig} \leftrightarrow N \leftrightarrow \ldots$

- $P_{orig} \rightarrow succ$ 仍然指向 N。它没有被修改为指向 x。
- `x->pred` 指向 `P_orig` (正确,来自初始化)。
- `x->succ` 指向 x (错误,自循环)。
- `N->pred` 指向 x (正确,来自第二步)。

结果:

- 新节点 x 被创建了。
- N 的前驱指针被正确地更新为 x。
- 但是，N 的原始前驱 P_{orig} 的后继指针没有被更新为 x (它仍然指向 N)。
- 更严重的是，新节点 x 的后继指针错误地指向了它自己 (`x->succ = x`)，而不是指向 N。

这意味着:

- 从 'P_{orig}' 向前遍历会跳过 'x' 直接到达 'N'。
- 从 'x' 向前遍历会陷入死循环。
- 从 'N' 向后遍历会到达 'x'，再从 'x' 向后遍历会到达 'P_{orig}'。这部分反向链接看起来是部分正确的 ($N \leftarrow x \leftarrow P_{orig}$)。

节点 'x' 没有被正确地插入到链表中，并且链表的原有结构（至少是 'x' 周围的链接）被破坏了。例如，'x' 形成了一个孤立的循环，并且 'P_{orig}' 和 'N' 之间的前向链接没有正确地包含 'x'。

因此:

- ***A.* 能正常插入节点**: 错误，链接不正确。
- ***B.* 不能插入节点，原列表仍然保持不变**: 错误，'*N->pred*' 被修改，新节点 'x' 被创建，列表结构已改变。
- ***C.* 不能插入节点，原列表的结构被破坏**: 正确。术语"不能插入节点"可以理解为"未能成功地、正确地插入节点"。由于错误的指针操作，列表的完整性和正确性被破坏了。
- ***D.* 能否插入节点与当时列表的结构有关**: 错误，这个逻辑缺陷是普遍的，不取决于列表的特定初始结构（如列表为空或只有一个节点等，只要 '*this*' 和 '*this->pred*' 是有效操作对象）。

44

> **我们可以考虑通过如下方式加快循秩访问的速度: 如果 r>n/2，则我们可以从尾部哨兵开始不断访问 pred()，最终从后向前地找到秩为 r 的节点。关于这种优化，哪种说法是错误的?**:
>
> ① 从期望的角度看，r 在 [0,n) 中是等概率分布的话，那么在循秩访问的过程中，对列表元素的访问次数可以节约一半。
>
> ② 原有方法访问最慢的情形大致出现在 r≈n 时，而改进后的方法访问最慢的情形大致出现在 r≈n/2 时。
>
> ③ 当对于列表的访问集中在列表尾部时，这种优化策略的效果最明显。
>
> ④ 通过这样的优化，我们可以使循秩访问时间复杂度优于 O(n)。

Solution 4. 正确答案为 D。

假设列表有 n 个元素，秩从 0 到 $n-1$。"循秩访问"指的是找到秩为 r 的节点。优化策略:

- 如果 $r \le n/2$ (或 $r < n/2$), 从头哨兵开始向前访问 $r+1$ 次 '*succ()*' (或者 r 次，取决于如何计数和实现)。
- 如果 $r > n/2$, 从尾哨兵开始向后访问 $n-1-r+1$ 次 '*pred()*' (即 $n-r$ 次)。

让我们分析各个选项:

- ***A.* 从期望的角度看，*r* 在 *[0,n)* 中是等概率分布的话，那么在循秩访问的过程中，对列表元素的访问次数可以节约一半。**
  - **原有方法** (只从头访问): 访问秩为 r 的节点需要大约 $r+1$ 次操作。如果 r 在 $[0, n-1]$ 中等概率分布，平均访问次数约为 $\sum_{i=0}^{n-1}(i+1)/n \approx n/2$。

  - **改进方法**:
    * 如果 $0 \le r < n/2$，访问次数约为 $r+1$。
    * 如果 $n/2 \le r < n$，访问次数约为 $(n-1-r)+1 = n-r$。

    在两种情况下,最大访问次数都约为 $n/2$。期望访问次数为 $\left(\sum_{i=0}^{\lfloor n/2 \rfloor -1}(i+1) + \sum_{i=\lfloor n/2 \rfloor}^{n-1}(n-i)\right)/n$. 这大约是 $n/4$。例如，如果 $n = 100$: 原有平均: ≈ 50 改进后平均: ≈ 25
  - 此说法是**正确的**。平均访问次数确实可以减半。
- ***B.* 原有方法访问最慢的情形大致出现在 *r≈n* 时，而改进后的方法访问最慢的情形大致出现在 *r≈n/2* 时。**
  - **原有方法**: 最慢的情况是访问最后一个元素 (秩 $n-1$)，需要约 n 次操作。
  - **改进方法**:
    * 如果从头访问，最慢是 $r \approx n/2 - 1$, 需要约 $n/2$ 次操作。
    * 如果从尾访问，最慢是 $r \approx n/2$, 需要约 $n - n/2 = n/2$ 次操作。

    所以，改进后最慢的情况是访问中间附近的元素，需要约 $n/2$ 次操作。
  - 此说法是**正确的**。
- ***C.* 当对于列表的访问集中在列表尾部时，这种优化策略的效果最明显。**
  - 如果访问集中在列表尾部 (e.g., $r \approx n-1$):
    * 原有方法: $\approx n$ 次操作。
    * 改进方法: 从尾部访问，≈ 1 或几次操作。

    节约了近 n 次操作，效果非常明显。
  - 如果访问集中在列表头部 (e.g., $r \approx 0$):
    * 原有方法: ≈ 1 或几次操作。
    * 改进方法: 从头部访问，≈ 1 或几次操作。

    效果不明显。
  - 如果访问集中在列表中部 (e.g., $r \approx n/2$):
    * 原有方法: $\approx n/2$ 次操作。
    * 改进方法: $\approx n/2$ 次操作。

    效果不明显。
  - 因此，当访问集中在尾部时，改进最大。
  - 此说法是**正确的**。
- ***D.* 通过这样的优化，我们可以使循秩访问时间复杂度优于 *O(n)*。**
  - 原有方法的循秩访问时间复杂度是 $O(n)$ (最坏情况)。
  - 改进后的方法，最坏情况的访问次数是 $n/2$ (当 $r \approx n/2$ 时)。
  - 时间复杂度 $O(n/2)$ 仍然是 $O(n)$。大 O 表示法忽略常数因子。
  - 虽然实际操作次数减少了,但渐进时间复杂度并未改变其类别。它没有变成 $O(\log n)$ 或 $O(1)$ 等优于 $O(n)$ 的复杂度。
  - 此说法是**错误的**。

因此，说法 D 是错误的。

45

> **有序列表唯一化算法的过程是:**:
>
> ① 只保留每个相等元素区间的第一个元素

> ② 每遇到一个元素，向后查找并删除与之雷同者
> ③ 每遇到一个元素，向前查找并删除与之雷同者
> ④ 检查 (n C 2) 个不同的元素对，对于每一对元素，若雷同则任意删除其一

Solution 5. 正确答案为 *A*。

有序列表唯一化（*uniquify*）算法的目的是移除重复的元素，使得每个值只出现一次，同时保持元素的相对顺序。由于列表是**有序的**，所有相等的元素必然是连续排列的，形成"相等元素区间"。

- ***A.* 只保留每个相等元素区间的第一个元素**
  - 这是对标准高效唯一化算法过程的准确描述。算法遍历列表，当遇到一个新的"相等元素区间"时，它会保留这个区间的第一个元素，并跳过或逻辑上移除该区间内所有后续的相同元素。
  - 例如，对于有序列表 *'L = 1, 2, 2, 2, 3, 3, 4'*:
    * 区间 *'1'*: 保留 *1.*
    * 区间 *'2, 2, 2'*: 保留第一个 *2*，跳过其余两个 *2.*
    * 区间 *'3, 3'*: 保留第一个 *3*，跳过其余一个 *3.*
    * 区间 *'4'*: 保留 *4.*
  - 结果为 *'1, 2, 3, 4'.*
  - 常见的实现方式（如数组的双指针法，或链表的逐个检查后继法）都遵循这一核心逻辑。
  - 此说法是**正确的**。
- ***B.* 每遇到一个元素，向后查找并删除与之雷同者**
  - 这个描述可以实现唯一化。对于有序列表，"向后查找"可以简化为检查紧随其后的元素。例如，在链表中，可以检查 *'current->data == current->next->data'*，如果相同则删除 *'current->next'*。
  - 虽然这是一个可行的过程，但选项 *A* 更准确地描述了处理"相等元素区间"的整体策略，这是有序性带来的关键结构。选项 *A* 描述了算法的目标和对这些区间的操作，而选项 *B* 描述了一种可能的、较低层次的迭代步骤。
  - 对于数组，如果"向后查找并删除"意味着对每个元素都进行一次独立的扫描和删除操作，可能会导致效率低下。
- ***C.* 每遇到一个元素，向前查找并删除与之雷同者**
  - 这个过程不太直观。如果算法从头到尾处理列表，并且目标是保留每组重复元素中的第一个，那么当遇到一个元素时，应该将其与"前一个被保留的唯一元素"进行比较。如果相同，则当前元素是重复的。向前查找并删除"与之雷同者"（即前一个元素）会违反"保留第一个"的常规。
  - 此说法描述的不是标准唯一化过程。
- ***D.* 检查 *(n C 2)* 个不同的元素对，对于每一对元素，若雷同则任意删除其一**
  - 这是非常低效的暴力方法，时间复杂度至少为 $O(n^2)$。它没有利用列表已排序的特性。
  - 标准的唯一化算法利用有序性，可以在 $O(n)$ 时间内完成。
  - 此说法是**错误的**。

综上，选项 *A* 最准确地描述了有序列表唯一化算法的核心过程和逻辑。

46

> **能否在有序列表中用二分查找使得时间复杂度降为 O($\log_2 n$)?**:
> ① 能
> ② 不能，因为列表扩容的分摊复杂度不是 O(1)
> ③ 不能，因为列表不能高效地循秩访问
> ④ 能，因为列表删除节点的时间复杂度为 O(1)

Solution 6. 正确答案为 C。

二分查找算法的核心思想是不断将搜索区间减半。为了有效地做到这一点，算法需要在每一步都能快速访问到当前搜索区间的中间元素。理想情况下，访问任意指定秩（索引）的元素的成本应为 $O(1)$。

- **对于基于数组的有序列表（如向量 *Vector*）**:
  - 元素在内存中是连续存储的。
  - 访问任意秩 r 的元素 *(e.g., 'array[r]')* 的时间复杂度是 $O(1)$。
  - 因此，在有序的基于数组的列表上，二分查找可以达到 $O(\log_2 n)$ 的时间复杂度。在这种情况下，选项 A"能"是成立的。
- **对于基于指针的有序列表（如链表 *LinkedList*）**:
  - 元素在内存中的位置可能不连续，通过指针链接。
  - 访问秩为 r 的元素需要从列表头部（或尾部）开始，沿着指针逐个遍历，直到到达第 r 个元素。这个操作的时间复杂度是 $O(r)$，最坏情况下是 $O(n)$。
  - 如果在链表上尝试实现二分查找，每次确定"中间"元素的秩后，定位到该元素本身就需要 $O(n)$（或 $O(\textit{current interval size})$）的时间。这使得总时间复杂度远高于 $O(\log_2 n)$（例如，可能是 $O(n)$ 或 $O(n \log n)$，取决于实现方式）。
  - 因此，对于链表,"列表不能高效地循秩访问"是正确的，这导致二分查找无法达到 $O(\log_2 n)$ 的复杂度。

分析选项:

- ***A.* 能**: 此说法仅对特定类型的列表（如基于数组的列表）成立，但对所有类型的"列表"(如链表）不成立。
- ***B.* 不能，因为列表扩容的分摊复杂度不是 *O(1)***: 列表扩容问题主要与动态数组（向量）相关，并且即使扩容分摊复杂度不是 $O(1)$（例如，如果是 $O(n)$ 的线性增长策略），这本身并不直接阻止在数据稳定后进行 $O(\log_2 n)$ 的二分查找（只要随机访问是 $O(1)$）。这不是二分查找可行性的核心原因。
- ***C.* 不能，因为列表不能高效地循秩访问**: 这是关键。如果"列表"泛指包括链表在内的数据结构，那么链表确实不能高效地（即 $O(1)$ 时间）循秩访问。这是导致标准二分查找在链表上效率低下的根本原因。因此，如果考虑到链表，那么二分查找不能保证 $O(\log_2 n)$ 的复杂度。
- ***D.* 能，因为列表删除节点的时间复杂度为 *O(1)***: 删除节点的复杂度与查找算法的复杂度是两回事。而且，虽然在已知节点位置的情况下，链表删除是 $O(1)$，但数组列表的删除通常是 $O(n)$。

结论: 题目问"能否...", 如果存在一种常见的列表类型使得二分查找无法达到 $O(\log_2 n)$，并且有一个选项能正确解释原因，那么这个选项就是答案。链表是列表的一种重要形式，它不支持高效的循秩访问。因此，由于列表（特指链表或不保证 $O(1)$ 循秩访问的列表）不能高效地循秩访问，二分查找的时间复杂度不能保证为 $O(\log_2 n)$。选项 C 准确地指出了这个核心障碍。

47

> **V={11, 5, 7, 13, 2, 3}，对 V 进行选择排序，被选为未排序子向量中最大的元素依次为: :**
>
> ① 11, 5, 7, 2, 3
>
> ② 13, 7, 11, 2, 5
>
> ③ 13, 11, 7, 5, 3
>
> ④ 2, 11, 13, 5, 7

Solution 7. 正确答案为 *C*。

选择排序（*Selection Sort*）的过程是：在每一轮中，从当前未排序的部分中选出最大（或最小）的元素，然后将其放置到已排序部分的末尾（或开头）。题目要求“被选为未排序子向量中最大的元素依次为”，这意味着我们每一轮都从未排序的部分找出最大的元素。

初始向量 *V = {11, 5, 7, 13, 2, 3}*。设 n 为向量的长度，即 $n = 6$. 我们将从未排序的子向量 *V[0...i]* 中选择最大元素，并将其与 *V[i]* 交换，然后缩小子向量的范围 *i* 从 $n - 1$ 减到 *1*。

- **第 *1* 轮** *(*未排序子向量 *V[0...5] = {11, 5, 7, 13, 2, 3}):*
  - 未排序部分为 *{11, 5, 7, 13, 2, 3}*。
  - 最大的元素是 ***13*** *(*位于索引 *3)*。
  - 将 *13* 与当前未排序部分的最后一个元素 *V[5] (*即 *3)* 交换。
  - *V* 变为 *{11, 5, 7, 3, 2,* ***13****}*。
  - 被选出的最大元素：*13*。
- **第 *2* 轮** *(*未排序子向量 *V[0...4] = {11, 5, 7, 3, 2}):*
  - 未排序部分为 *{11, 5, 7, 3, 2}*。
  - 最大的元素是 ***11*** *(*位于索引 *0)*。
  - 将 *11* 与当前未排序部分的最后一个元素 *V[4] (*即 *2)* 交换。
  - *V* 变为 *{2, 5, 7, 3,* ***11****, 13}*。
  - 被选出的最大元素：*11*。
- **第 *3* 轮** *(*未排序子向量 *V[0...3] = {2, 5, 7, 3}):*
  - 未排序部分为 *{2, 5, 7, 3}*。
  - 最大的元素是 ***7*** *(*位于索引 *2)*。
  - 将 *7* 与当前未排序部分的最后一个元素 *V[3] (*即 *3)* 交换。
  - *V* 变为 *{2, 5, 3, 7, 11, 13}*。
  - 被选出的最大元素：*7*。
- **第 *4* 轮** *(*未排序子向量 *V[0...2] = {2, 5, 3}):*
  - 未排序部分为 *{2, 5, 3}*。
  - 最大的元素是 ***5*** *(*位于索引 *1)*。
  - 将 *5* 与当前未排序部分的最后一个元素 *V[2] (*即 *3)* 交换。
  - *V* 变为 *{2, 3,* ***5****, 7, 11, 13}*。
  - 被选出的最大元素：*5*。
- **第 *5* 轮** *(*未排序子向量 *V[0...1] = {2, 3}):*
  - 未排序部分为 *{2, 3}*。
  - 最大的元素是 ***3*** *(*位于索引 *1)*。
  - 将 *3* 与当前未排序部分的最后一个元素 *V[1] (*即 *3)* 交换。*(*元素已在正确位置，无需实际移

动)

- *V* 变为 *{2, **3**, 5, 7, 11, 13}*。
- 被选出的最大元素：*3*。

经过 $n-1=5$ 轮后，向量排序完毕。被选为未排序子向量中最大的元素依次为：*13, 11, 7, 5, 3*。

这与选项 *C* 相符。

48

> **为了保证 selectSort() 算法的稳定性，我们采取的措施是: :**
> ① selectMax() 中对于多个相等的最大元素，选取其中位置最靠后者
> ② selectMax() 中对于多个相等的最大元素，选取其中位置最靠前者
> ③ 先调用 deduplicate() 删除所有重复元素
> ④ 无论实现细节如何，该算法本来就是稳定的

Solution 8. 正确答案为 *A*。

选择排序（*Selection Sort*）本身通常被认为是不稳定的排序算法。稳定性是指如果排序前有两个相等的元素，并且元素 *A* 在元素 *B* 之前，那么排序后元素 *A* 仍然在元素 *B* 之前。选择排序在交换元素时可能会破坏这种相对顺序。

题目中提到了 *‘selectMax()‘*，这暗示了选择排序的一种实现方式：在每一轮中，从未排序的部分选择最大的元素，并将其放置到未排序部分的末尾（或者说，已排序部分的开始，如果已排序部分从数组末尾开始增长）。具体来说，假设我们从未排序的子向量 *‘V[0...i]‘* 中选择最大元素，并将其与 *‘V[i]‘* 交换，其中 *‘i‘* 从 *‘n-1‘* 递减到 *‘1‘*。

- ***D.* 无论实现细节如何，该算法本来就是稳定的**：这是错误的。标准的选择排序是不稳定的。例如，如果数组是 *‘(3,a), (2,b), (3,c)‘* (值, 原始标识)，目标是升序排序。如果采用“从未排序部分选最小，放到最前面”:
  - 第 *1* 轮：最小的是 *‘(2,b)‘*。与 *‘(3,a)‘* 交换。数组变为 *‘(2,b), (3,a), (3,c)‘*。
  - 第 *2* 轮：从未排序的 *‘(3,a), (3,c)‘* 中选最小。如果选 *‘(3,a)‘* (第一个遇到的)，与自身交换。数组 *‘(2,b), (3,a), (3,c)‘*。稳定。
  - 但如果 *‘selectMin‘* 的实现导致它可能选到 *‘(3,c)‘*（例如，如果它从后向前扫描找最小值），然后与 *‘(3,a)‘* 交换，就会变成 *‘(2,b), (3,c), (3,a)‘*，不稳定。

  所以，实现细节很重要。
- ***C.* 先调用 *deduplicate()* 删除所有重复元素**：这会改变原问题。虽然消除了重复元素后稳定性的概念对于这些特定值不再适用，但这并不是保证原 *‘selectSort()‘* 算法稳定性的措施。
- 现在考虑 *‘selectMax()‘* 的版本，将最大元素放到当前未排序部分的末尾 *‘V[i]‘*。设想数组 *‘V = (X, item1), (Y, item2), (X, item3)‘*，其中 *‘item1‘* 在 *‘item3‘* 之前，且 *‘X‘* 是最大值。我们要将最大的元素放到 *‘V[i]‘*。
  - ***A. selectMax()* 中对于多个相等的最大元素，选取其中位置最靠后者**：“位置最靠后者”意味着在扫描范围 *‘V[0...i]‘* 中，如果多个元素都是最大值，选择索引最大的那个。例如，*‘V = (X,1), (X,2)‘*。*‘i=1‘*。未排序部分 *‘V[0...1] = (X,1), (X,2)‘*。*‘selectMax‘* 扫描 *‘V[0...1]‘*。最大元素有 *‘(X,1)‘* (索引 *0*) 和 *‘(X,2)‘* (索引 *1*)。选取“位置最靠后者”，即 *‘(X,2)‘* (索引 *1*)。将其与 *‘V[i]=V[1]‘* (即 *‘(X,2)‘* 自身) 交换。数组仍为 *‘(X,1), (X,2)‘*。下一轮 *‘i=0‘*。未排序 *‘V[0...0] = (X,1)‘*。选 *‘(X,1)‘* 与 *‘V[0]‘* 交换。最终数组 *‘(X,1), (X,2)‘*。稳定。

– ***B. selectMax()* 中对于多个相等的最大元素，选取其中位置最靠前者**: 例如，*'V = (X,1), (X,2)'*。*'i=1'*。未排序部分 *'V[0...1] = (X,1), (X,2)'*。*'selectMax'* 扫描 *'V[0...1]'*。最大元素有 *'(X,1)'* (索引 *0*) 和 *'(X,2)'* (索引 *1*)。选取"位置最靠前者", 即 *'(X,1)'* (索引 *0*)。将其与 *'V[i]=V[1]'* (即 *'(X,2)'*) 交换。数组变为 *'(X,2), (X,1)'*。最终数组 *'(X,2), (X,1)'*。不稳定。

因此，为了在这种"选最大放到末尾"的选择排序中保证稳定性，当 *'selectMax()'* 从 *'V[0...i]'* 中找到最大元素时，如果存在多个相等的最大元素，必须选择它们中索引最大的那个（即"位置最靠后者"）。这样，当它与 *'V[i]'* 交换时，它不会越过其他与它相等的、且原本在它之后但在 *'V[i]'* 之前的元素。所以，措施 *A* 可以保证这种特定实现的 *'selectSort()'* 算法的稳定性。

49

> **对于规模为 n 的向量或列表，选择排序和冒泡排序的最坏时间复杂度为: :**
> ① $\Theta(n\log_2 n), \Theta(n^2)$
> ② $\Theta(n\log_2 n), \Theta(n\log_2 n)$
> ③ $\Theta(n^2), \Theta(n^2)$
> ④ $\Theta(n^2), \Theta(n\log_2 n)$

Solution 9. 正确答案为 *C*。

- **选择排序 *(Selection Sort)*:**
  – 无论输入数据的初始顺序如何，选择排序都需要进行 $n-1$ 轮。
  – 在第 k 轮中，它需要从未排序的 $n-k+1$ 个元素中找到最小（或最大）的元素。这需要 $n-k$ 次比较。
  – 总的比较次数大约是 $(n-1)+(n-2)+\ldots+1 = n(n-1)/2$。
  – 因此，选择排序的最好、平均和最坏时间复杂度都是 $\Theta(n^2)$。
- **冒泡排序 *(Bubble Sort)*:**
  – 在最坏情况下（例如，列表完全逆序），冒泡排序需要进行 $n-1$ 趟。
  – 在第 k 趟中，它会进行 $n-k$ 次相邻元素的比较和可能的交换。
  – 总的比较次数在最坏情况下是 $(n-1)+(n-2)+\ldots+1 = n(n-1)/2$。
  – 因此，冒泡排序的最坏时间复杂度是 $\Theta(n^2)$。
  – *(*注：冒泡排序的最好情况时间复杂度，如果使用标志位优化，可以达到 $\Theta(n)$，但题目问的是最坏情况。*)*

所以，对于规模为 n 的向量或列表，选择排序和冒泡排序的最坏时间复杂度均为 $\Theta(n^2)$。

50

> **n 个元素的序列所含逆序对的个数最大是: :**
> ① n!
> ② n!/2
> ③ (n(n-1))/2
> ④ n

Solution 10. 正确答案为 C。

一个逆序对是指在序列中，一对元素 (a_i, a_j) 满足 $i < j$ 但 $a_i > a_j$。逆序对的个数在序列完全逆序（即从大到小排列）时达到最大。

考虑一个包含 n 个不同元素的序列。当它完全逆序排列时，例如 $\{e_n, e_{n-1}, \ldots, e_2, e_1\}$ 其中 $e_n > e_{n-1} > \ldots > e_1$。

- 第一个元素 e_n 与后面所有 $n-1$ 个元素都构成逆序对。
- 第二个元素 e_{n-1} *(在原序列中)* 与后面所有 $n-2$ 个元素都构成逆序对。
- ...
- 第 $n-1$ 个元素 e_2 与最后一个元素 e_1 构成 *1* 个逆序对。
- 最后一个元素 e_1 不与任何后续元素构成逆序对。

所以，最大逆序对的个数是 $(n-1)+(n-2)+\ldots+1+0$. 这是一个等差数列的和，其值为 $\frac{(n-1)\times((n-1)+1)}{2} = \frac{(n-1)n}{2} = \frac{n(n-1)}{2}$。

这也可以理解为从 n 个元素中选取任意两个元素组成一对，这样的组合有 $C(n, 2)$ 种。在完全逆序的序列中，每一对这样的组合都构成一个逆序对。$C(n, 2) = \binom{n}{2} = \frac{n!}{2!(n-2)!} = \frac{n(n-1)}{2}$。

因此，n 个元素的序列所含逆序对的个数最大是 $\frac{n(n-1)}{2}$。

51

insertionSort() 的平均、最坏时间复杂度分别为：:

① $\Theta(n), \Theta(n^2)$
② $\Theta(n^2), \Theta(n^2)$
③ $\Theta(n \log_2 n), \Theta(n^2)$
④ $\Theta(n \log_2 n), \Theta(n \log_2 n)$

Solution 11. 正确答案为 B。

插入排序 *(Insertion Sort)*: 插入排序通过构建有序序列，对于未排序数据，在已排序序列中从后向前扫描，找到相应位置并插入。

- **最坏时间复杂度**:
  - 当输入数组完全逆序时，发生最坏情况。
  - 对于第 i 个元素（从第二个元素开始，即 $i = 1$ 到 $n-1$），它需要与前面所有 i 个已排序的元素进行比较，并可能需要将这 i 个元素都向后移动一位。
  - 总的比较和移动次数大约是 $1+2+\ldots+(n-1) = \frac{n(n-1)}{2}$。
  - 因此，最坏时间复杂度是 $\Theta(n^2)$。
- **平均时间复杂度**:
  - 在平均情况下（例如，输入数组是随机排列的），对于第 i 个元素，我们期望它平均需要与前面 $i/2$ 个元素进行比较和移动。
  - 总的比较和移动次数仍然是 $O(n^2)$。具体来说，是 $\sum_{i=1}^{n-1} \frac{i}{2} \approx \frac{n^2}{4}$。
  - 因此，平均时间复杂度是 $\Theta(n^2)$。
- **最好时间复杂度**:
  - 当输入数组已经排序时，发生最好情况。
  - 对于第 i 个元素，只需要与前一个元素比较一次即可确定其位置（不需要移动）。
  - 总的比较次数是 $n-1$。

– 因此，最好时间复杂度是 $\Theta(n)$。

题目要求平均和最坏时间复杂度，它们分别为 $\Theta(n^2)$ 和 $\Theta(n^2)$。

52

> **对于插入过程排序中的已排序子序列（设其长度为 k）: :**
> ① 其中的元素是整个序列中最小的 k 个元素
> ② 其中的元素是整个序列中最大的 k 个元素
> ③ 其中的元素是原序列中位于前方的 k 个元素
> ④ 其中的元素是原序列中位于后方的 k 个元素

Solution 12. 正确答案为 *C*。

插入排序（*Insertion Sort*）的工作原理是逐步构建一个已排序的子序列。在算法的第 k 步（或者说，当已排序子序列的长度为 k 时），这个已排序的子序列是由原序列中最初的前 k 个元素组成的，只是它们现在已经被排好序了。

例如，考虑序列 *V = {5, 2, 4, 1, 3}*:

- **初始**: 已排序子序列为空。
- **处理第一个元素 (5)**: 已排序子序列为 *{5}*。长度 $k = 1$。它由原序列的第一个元素组成。
- **处理第二个元素 (2)**: 将 *2* 插入到 *{5}* 中。已排序子序列为 *{2, 5}*。长度 $k = 2$。它由原序列的前两个元素 *{5, 2}* 排序后得到。
- **处理第三个元素 (4)**: 将 *4* 插入到 *{2, 5}* 中。已排序子序列为 *{2, 4, 5}*。长度 $k = 3$。它由原序列的前三个元素 *{5, 2, 4}* 排序后得到。
- **处理第四个元素 (1)**: 将 *1* 插入到 *{2, 4, 5}* 中。已排序子序列为 *{1, 2, 4, 5}*。长度 $k = 4$。它由原序列的前四个元素 *{5, 2, 4, 1}* 排序后得到。
- **处理第五个元素 (3)**: 将 *3* 插入到 *{1, 2, 4, 5}* 中。已排序子序列为 *{1, 2, 3, 4, 5}*。长度 $k = 5$。它由原序列的前五个元素 *{5, 2, 4, 1, 3}* 排序后得到。

分析选项:

- ***A.* 其中的元素是整个序列中最小的 *k* 个元素**: 错误。例如，当 $k = 1$ 时，已排序子序列是 *{5}*，但 *5* 不是整个序列中最小的元素。
- ***B.* 其中的元素是整个序列中最大的 *k* 个元素**: 错误。
- ***C.* 其中的元素是原序列中位于前方的 *k* 个元素**: 正确。已排序子序列总是由原序列中从第 *0* 个到第 $k - 1$ 个元素（即前 k 个元素）经过排序后组成的。
- ***D.* 其中的元素是原序列中位于后方的 *k* 个元素**: 错误。

53

> **在插入排序的某一步后得到如下子序列 V={2, 7, 13, 5, 3}，此时已排序部分有 3 个元素。经过又一轮迭代后的结果是: :**
> ① $\{2, 3, 7, 13, 5\}$
> ② $\{2, 7, 13, 3, 5\}$
> ③ $\{2, 5, 7, 13, 3\}$

> ④ {3, 2, 7, 13, 5}

Solution 13. 正确答案为 *C*。

当前序列 *V = {2, 7, 13, 5, 3}*。已排序部分有 *3* 个元素，这意味着 *V[0...2] = {2, 7, 13}* 是已排序的。未排序部分的第一个元素是 *V[3] = 5*。

下一轮迭代的任务是将元素 *5* 插入到已排序部分 *{2, 7, 13}* 中的正确位置。

(1) 取出元素 *‘e = 5‘*。

(2) 将 *‘e‘* 与已排序部分的最后一个元素 *‘V[2] = 13‘* 比较。

- *‘5 < 13‘*，所以 *‘13‘* 需要向右移动。
- *V* 变为 *{2, 7, _, 13, 3}* (逻辑上，*5* 被暂存，*V[2]* 的位置空出，*V[3]* 的值变为 *13)*。

(3) 将 *‘e‘* 与已排序部分的前一个元素 *‘V[1] = 7‘* 比较。

- *‘5 < 7‘*，所以 *‘7‘* 需要向右移动。
- *V* 变为 *{2, _, 7, 13, 3}* (逻辑上，*V[1]* 的位置空出，*V[2]* 的值变为 *7)*。

(4) 将 *‘e‘* 与已排序部分的前一个元素 *‘V[0] = 2‘* 比较。

- *‘5 > 2‘*，所以 *‘5‘* 应该插入到 *‘2‘* 的后面，即当前空出的位置 *V[1]*。

(5) 将 *‘e=5‘* 放入 *V[1]*。

经过这一轮迭代后，序列 *V* 变为 *{2, 5, 7, 13, 3}*。已排序部分现在是 *{2, 5, 7, 13}*，未排序部分是 *{3}*。

这与选项 *C* 相符。

54

> **下列关于向量和列表的说法错误的是:** :
>
> ① 向量通常在内存中占据连续的空间，列表则通常不是如此
>
> ② 在有序向量中查找渐进地比在有序列表中查找快
>
> ③ 向量归并排序的时间复杂度是 $\mathrm{O}(\mathrm{nlog_2 n})$，而列表为 $\Omega(n^2)$
>
> ④ 列表删除单个节点渐进地比向量删除单个元素快

Solution 14. 正确答案为 *C*。

- ***A.* 向量通常在内存中占据连续的空间，列表则通常不是如此**
  - **向量 *(Vector)***，如 *C++* 中的 *‘std::vector‘* 或动态数组，其元素在内存中是连续存储的。这允许通过索引进行快速的随机访问。
  - **列表 *(List)***，特指链表 *(Linked List)*，其节点在内存中的位置通常是不连续的，节点之间通过指针链接。
  - 此说法是**正确的**。
- ***B.* 在有序向量中查找渐进地比在有序列表中查找快**
  - **有序向量**：由于支持 $O(1)$ 的随机访问（循秩访问），可以使用二分查找，时间复杂度为 $O(\log n)$。
  - **有序列表 *(*链表*)***：即使列表有序，要访问中间元素也需要从头或尾开始遍历，循秩访问的时间复杂度是 $O(n)$。因此，在链表上直接应用标准的二分查找效率不高。通常在有序链表上的查找是线性查找，时间复杂度为 $O(n)$。
  - $O(\log n)$ 渐进地快于 $O(n)$。
  - 此说法是**正确的**。

- ***C.* 向量归并排序的时间复杂度是 $O(n\log_2 n)$，而列表为** $\Omega(n^2)$
  - **向量归并排序**：归并排序对数组（向量）的时间复杂度是 $O(n\log n)$。
  - **列表归并排序**：归并排序对链表同样可以实现 $O(n\log n)$ 的时间复杂度。链表的合并操作 (*merge*) 可以通过修改指针完成，效率很高。将链表分成两半也可以在 $O(n)$ 时间内完成（或通过更巧妙的方式在递归中自然实现）。
  - 因此，声称列表归并排序的时间复杂度为 $\Omega(n^2)$ *(*意味着其最少也是 n^2 级别*)* 是错误的。列表的归并排序也是高效的。
  - 此说法是**错误的**。
- ***D.* 列表删除单个节点渐进地比向量删除单个元素快**
  - **列表 *(*链表*)* 删除**：如果已知要删除的节点（或其前驱节点，对于单向链表），删除操作仅涉及修改指针，时间复杂度为 $O(1)$。
  - **向量删除**：从向量中删除一个元素（除非是末尾元素），需要将后续所有元素向前移动以填补空位，这个操作的时间复杂度是 $O(n)$（其中 n 是被移动元素的数量，最坏情况下是整个向量的长度）。
  - 即使考虑到查找元素的时间，如果题目暗示的是"一旦定位到元素/节点后"的删除操作，那么列表的 $O(1)$ 确实比向量的 $O(n)$ 快。如果题目泛指整个删除过程（查找 + 删除），对于无序结构查找都是 $O(n)$，但删除操作本身列表更快。
  - 此说法通常被认为是**正确的**，强调的是删除操作本身的效率。

因此，说法 C 是错误的。

55

> **为了在列表中插入一个新节点 node 作为 p 的直接前驱，有四个相关的语句 1.p->pred->succ = node 2.node->pred = p->pred 3.node->succ = p 4.p->pred = node 上述语句执行顺序正确的是: :**
>
> A 3241
>
> B 3214
>
> C 1432
>
> D 4231

Solution 15. 正确答案为 B。

假设原始列表片段为 *‘... <- A <-> p -> ...‘*，其中 *‘A‘* 是 *‘p‘* 的原始直接前驱，即 *‘A = p->pred‘*。我们的目标是插入 *‘node‘* 使得列表变为 *‘... <- A <-> node <-> p -> ...‘*。

这需要进行以下链接调整:

(1) *‘node‘* 的后继应指向 *‘p‘ (‘node->succ = p‘)*。

(2) *‘node‘* 的前驱应指向 *‘A‘ (‘node->pred = A‘)*。

(3) *‘A‘* 的后继应指向 *‘node‘ (‘A->succ = node‘)*。

(4) *‘p‘* 的前驱应指向 *‘node‘ (‘p->pred = node‘)*。

让我们分析给出的语句:

- *‘1 p->pred->succ = node‘*: 这相当于 *‘A->succ = node‘*，前提是 *‘p->pred‘* 此时仍然指向 *‘A‘*。
- *‘2 node->pred = p->pred‘*: 这相当于 *‘node->pred = A‘*，前提是 *‘p->pred‘* 此时仍然指向 *‘A‘*。
- *‘3 node->succ = p‘*: 设置 *‘node‘* 的后继。
- *‘4 p->pred = node‘*: 设置 *‘p‘* 的前驱为新节点 *‘node‘*。

关键在于，语句 *1* 和 *2* 读取 *‘p->pred‘* 的值（即 *‘A‘*）。因此，它们必须在语句 *4*（即 *‘p->pred = node‘*，它会修改 *‘p->pred‘* 的值）执行之前执行。

让我们分析选项 *B: 3214*

(1) 3 ‘node->succ = p‘

- *‘node‘* 的 *‘succ‘* 指针指向 *‘p‘.*
- *‘node: [pred=?, data=?, succ=p]‘*

(2) 2 ‘node->pred = p->pred‘

- 此时 *‘p->pred‘* 仍然是原始的 *‘A‘.*
- *‘node‘* 的 *‘pred‘* 指针指向 *‘A‘.*
- *‘node: [pred=A, data=?, succ=p]‘*
- 此时 *‘node‘* 正确地链接了它的前驱和后继：*‘A <- node -> p‘.*
- 但是 *‘A‘* 的 *‘succ‘* 仍然指向 *‘p‘*，*‘p‘* 的 *‘pred‘* 仍然指向 *‘A‘.*

(3) 1 ‘p->pred->succ = node‘

- 此时 *‘p->pred‘* 仍然是原始的 *‘A‘.*
- 所以这条语句是 *‘A->succ = node‘.*
- *‘A‘* 的 *‘succ‘* 指针现在指向 *‘node‘.*
- 链接变为*: ‘A <-> node -> p‘. (‘p‘* 的 *‘pred‘* 仍然指向 *‘A‘)*

(4) 4 ‘p->pred = node‘

- *‘p‘* 的 *‘pred‘* 指针现在指向 *‘node‘.*
- 链接最终变为*: ‘A <-> node <-> p‘.*

这个顺序是正确的。

为什么其他选项是错误的:

- ***A. 3241****:* 在执行 *4 (‘p->pred = node‘)* 之后，*‘p->pred‘* 变成了 *‘node‘*。那么接下来的 □ *(‘p->pred->succ = node‘)* 就会变成 *‘node->succ = node‘*，这是错误的，它会使 *‘node‘* 指向自身，并覆盖在步骤 □ 中设置的 *‘node->succ = p‘*。
- ***C. 1432****:* 在执行 *4 (‘p->pred = node‘)* 之后，*‘p->pred‘* 变成了 *‘node‘*。那么接下来的 □ *(‘node->pred = p->pred‘)* 就会变成 *‘node->pred = node‘*，这是错误的。
- ***D. 4231****:* 在执行 *4 (‘p->pred = node‘)* 之后，*‘p->pred‘* 变成了 *‘node‘*。那么接下来的 □ *(‘node->pred = p->pred‘)* 就会变成 *‘node->pred = node‘*，这是错误的。

56

在有序列表中查找一个元素的时间复杂度是: :

① $\Omega(n \log_2 n)$

② $\Omega(n)$

③ $\mathrm{O}(\log_2 n)$

④ $\mathrm{O}(1)$

Solution 16. 正确答案为 *B*。

这个问题的答案取决于“列表”的具体实现（例如，是基于数组的列表还是基于链表的列表）。

- **如果“有序列表”是基于数组的 *(*如有序向量*)*:**
  - 由于数组支持高效的随机访问（循秩访问时间复杂度为 *O(1)*），我们可以使用二分查找。

  - 二分查找的时间复杂度是 $O(\log_2 n)$。
  - 在这种情况下，选项 C ($O(\log_2 n)$) 将是正确的。
- **如果“有序列表”是基于链表的**:
  - 链表不支持高效的随机访问。要访问链表中的第 k 个元素，通常需要从头节点开始遍历 k 步，时间复杂度为 $O(k)$。
  - 因此，在链表上直接应用标准的二分查找无法达到 $O(\log_2 n)$ 的效率。
  - 在有序链表中查找元素通常需要进行线性扫描，从头到尾（或直到找到元素或确定元素不存在）。
  - 线性扫描的时间复杂度在最坏情况下（例如，元素在末尾或不存在）是 $O(n)$，平均情况下也是 $O(n)$。因此，其时间复杂度为 $\Theta(n)$。
  - 如果时间复杂度是 $\Theta(n)$，那么它既是 $O(n)$ 也是 $\Omega(n)$。

考虑到在类似题目（如题目 *46* 和题目 *54* 的上下文）中，“列表”通常被用来指代不支持高效随机访问的数据结构（如链表），与“向量”（通常指数组）形成对比。题目 *46* 明确指出列表因不能高效循秩访问而不适用于 $O(\log_2 n)$ 的二分查找。题目 *54* 也指出向量查找比列表查找快。

基于此上下文，我们假设这里的“有序列表”指的是有序链表或类似的不能进行 $O(1)$ 循秩访问的结构。在这种情况下:

- 查找操作的时间复杂度是 $\Theta(n)$。
- 这意味着算法的运行时间 $T(n)$ 满足 $c_1 n \le T(n) \le c_2 n$ 对于足够大的 n。
- 因此，$T(n)$ 是 $O(n)$ 并且 $T(n)$ 是 $\Omega(n)$。

现在看选项:

- *A.* $\Omega(n\log_2 n)$*:* 对于线性扫描来说太高了。
- *B.* $\Omega(n)$*:* 如果复杂度是 $\Theta(n)$，那么 $\Omega(n)$ 是正确的。它表示在最坏情况下，算法至少需要线性时间。
- *C.* $O(\log_2 n)$*:* 这对于链表来说通常是不正确的。
- *D.* $O(1)$*:* 除非在非常特殊的情况下（例如，查找的元素总是在列表的开头），否则不正确。

由于对于有序链表的查找，其时间复杂度是 $\Theta(n)$，因此它必然是 $\Omega(n)$。选项 *B* 表明，在（链式）有序列表中查找元素，其时间复杂度的下界是线性的。这是因为在最坏情况下，可能需要检查所有 n 个元素。

因此，在将“列表”理解为链表（基于上下文）的前提下，$\Omega(n)$ 是对查找时间复杂度的正确描述（作为其下界）。

57

> **对列表 {11, 5, 7, 13, 2, 3} 进行选择排序，每一次 selectMax() 被选为未排序子列表中最大者的元素依次为: :**
>
> ① 11, 5, 7, 2, 3
> ② 13, 7, 11, 2, 5
> ③ 13, 11, 7, 5, 3
> ④ 2, 11, 13, 5, 7

Solution 17. 正确答案为 *C*。

这道题目与第 *47* 题完全相同。选择排序（*Selection Sort*）的过程是: 在每一轮中，从当前未排序的部分中选出最大（或最小）的元素，然后将其放置到已排序部分的末尾（或开头）。题目要求“每一次 *selectMax()* 被选为未排序子列表中最大者的元素依次为”。

初始列表 $V = \{11, 5, 7, 13, 2, 3\}$。设 n 为列表的长度，即 $n = 6$. 我们将从未排序的子列表 $V[0...i]$ 中选择最大元素，并将其与 $V[i]$ 交换，然后缩小子列表的范围 i 从 $n-1$ 减到 1。

- **第 1 轮** (未排序子列表 $V[0...5] = \{11, 5, 7, 13, 2, 3\}$):
  - 未排序部分为 $\{11, 5, 7, 13, 2, 3\}$。
  - 最大的元素是 ***13***。
  - 将 13 与当前未排序部分的最后一个元素 $V[5]$ (即 3) 交换。
  - V 变为 $\{11, 5, 7, 3, 2, \mathbf{13}\}$。
  - 被选出的最大元素：13。
- **第 2 轮** (未排序子列表 $V[0...4] = \{11, 5, 7, 3, 2\}$):
  - 未排序部分为 $\{11, 5, 7, 3, 2\}$。
  - 最大的元素是 ***11***。
  - 将 11 与当前未排序部分的最后一个元素 $V[4]$ (即 2) 交换。
  - V 变为 $\{2, 5, 7, 3, \mathbf{11}, 13\}$。
  - 被选出的最大元素：11。
- **第 3 轮** (未排序子列表 $V[0...3] = \{2, 5, 7, 3\}$):
  - 未排序部分为 $\{2, 5, 7, 3\}$。
  - 最大的元素是 7。
  - 将 7 与当前未排序部分的最后一个元素 $V[3]$ (即 3) 交换。
  - V 变为 $\{2, 5, 3, \mathbf{7}, 11, 13\}$。
  - 被选出的最大元素：7。
- **第 4 轮** (未排序子列表 $V[0...2] = \{2, 5, 3\}$):
  - 未排序部分为 $\{2, 5, 3\}$。
  - 最大的元素是 ***5***。
  - 将 5 与当前未排序部分的最后一个元素 $V[2]$ (即 3) 交换。
  - V 变为 $\{2, 3, \mathbf{5}, 7, 11, 13\}$。
  - 被选出的最大元素：5。
- **第 5 轮** (未排序子列表 $V[0...1] = \{2, 3\}$):
  - 未排序部分为 $\{2, 3\}$。
  - 最大的元素是 ***3***。
  - 将 3 与当前未排序部分的最后一个元素 $V[1]$ (即 3) 交换。(元素已在正确位置，无需实际移动)
  - V 变为 $\{2, \mathbf{3}, 5, 7, 11, 13\}$。
  - 被选出的最大元素：3。

经过 $n - 1 = 5$ 轮后，列表排序完毕。被选为未排序子列表中最大者的元素依次为：13, 11, 7, 5, 3。

这与选项 C 相符。

58

> **selectionSort() 算法的哪种实现是稳定的：:**
>
> ① 每一趟将最小元素移到前方，对于多个相等的最小元素，选取其中位置最靠前者。
>
> ② 每一趟将最大元素移到后方，对于多个相等的最大元素，选取其中位置最靠前者。
>
> ③ 每一趟将最小元素移到前方，对于多个相等的最小元素，选取其中位置最靠后者。

> ④ 以上实现皆稳定。

Solution 18. 正确答案为 A。

稳定性是指在排序过程中，具有相同键值的元素的相对顺序在排序后保持不变。

- ***A.* 每一趟将最小元素移到前方，对于多个相等的最小元素，选取其中位置最靠前者。**
  - 考虑序列 S = {…，(X，$item_i$)，…，(Y，$item_j$)，…(X，$item_k$)，…} *where* $item_i$ *is before* $item_k$ *in the original sequence.*
  - 当我们将最小元素放到已排序部分的末尾（即当前处理位置的前方）时，如果遇到多个相等的最小元素，选择它们中在当前未排序子序列里最靠前的那个（即原始索引最小的那个）。
  - 假设我们正在填充位置 p。我们在 $S[p \ldots n-1]$ 中寻找最小元素。
  - 如果 $(X, item_i)$ 和 $(X, item_k)$ 都是最小元素，且 $item_i$ 在 $item_k$ 之前。如果我们选择 $(X, item_i)$ *(*最靠前者*)* 并将其与 $S[p]$ 交换，那么 $(X, item_i)$ 就被放到了正确的位置。之后处理 $(X, item_k)$ 时，它不会被错误地移动到 $(X, item_i)$ 之前。
  - 这种方法是**稳定的**。例如，对于 *'(3,a), (2,b), (3,c)':*
    - ***(1)*** 找到最小 *'(2,b)'*。交换 *'(3,a)'* 和 *'(2,b)'* → *'(2,b), (3,a), (3,c)'.*
    - ***(2)*** 在 *'(3,a), (3,c)'* 中找最小。两者相等。选最靠前者 *'(3,a)'*。与自身交换 *(*无变化*)*。→ *'(2,b), (3,a), (3,c)'.*
  - 结果 *'(2,b), (3,a), (3,c)'* 是稳定的。
- ***B.* 每一趟将最大元素移到后方，对于多个相等的最大元素，选取其中位置最靠前者。**
  - 考虑序列 *'S = (3,a), (3,b)' (*目标是升序，所以 *'(3,a)'* 应该在 *'(3,b)'* 之前*)*。
  - 我们将最大元素放到未排序部分的末尾。
  - 第 *1* 趟 *(*处理 *'S[0...n-1]'*，将最大者放到 *'S[n-1]')*: 未排序部分 *'(3,a), (3,b)'*。最大元素是 *'3'*。有两个：*'(3,a)'* 和 *'(3,b)'*。选取"位置最靠前者"，即 *'(3,a)'*。将 *'(3,a)'* 与 *'S[n-1]' (*即 *'(3,b)')* 交换。序列变为 *'(3,b), (3,a)'*。
  - 此时，*'(3,b)'* 在 *'(3,a)'* 之前，原始顺序被破坏。
  - 这种方法是**不稳定的**。
- ***C.* 每一趟将最小元素移到前方，对于多个相等的最小元素，选取其中位置最靠后者。**
  - 考虑序列 *'S = (3,a), (3,b)'*。
  - 我们将最小元素放到未排序部分的前方。
  - 第 *1* 趟 *(*处理 *'S[0...n-1]'*，将最小者放到 *'S[0]')*: 未排序部分 *'(3,a), (3,b)'*。最小元素是 *'3'*。有两个：*'(3,a)'* 和 *'(3,b)'*。选取"位置最靠后者"，即 *'(3,b)'*。将 *'(3,b)'* 与 *'S[0]' (*即 *'(3,a)')* 交换。序列变为 *'(3,b), (3,a)'*。
  - 此时，*'(3,b)'* 在 *'(3,a)'* 之前，原始顺序被破坏。
  - 这种方法是**不稳定的**。
- ***D.* 以上实现皆稳定。**
  - 由于 B 和 C 不稳定，此选项错误。

因此，只有选项 A 描述的实现是稳定的。

59

> **对于插入排序过程中的已排序子序列（设其长度为 k）::**
> ① 其中的元素是整个序列中最小的 k 个元素
> ② 其中的元素是整个序列中最大的 k 个元素
> ③ 其中的元素是原序列中位于前方的 k 个元素
> ④ 其中的元素是原序列中位于后方的 k 个元素

Solution 19. 正确答案为 *C*。

这道题目与第 *52* 题完全相同。插入排序（*Insertion Sort*）的工作原理是逐步构建一个已排序的子序列。在算法的第 k 步（或者说，当已排序子序列的长度为 k 时），这个已排序的子序列是由原序列中最初的前 k 个元素组成的，只是它们现在已经被排好序了。

例如，考虑序列 *V = {5, 2, 4, 1, 3}*:

- ***初始***: 已排序子序列为空。
- **处理第一个元素 *(5)***: 已排序子序列为 *{5}*。长度 $k = 1$。它由原序列的第一个元素组成。
- **处理第二个元素 *(2)***: 将 *2* 插入到 *{5}* 中。已排序子序列为 *{2, 5}*。长度 $k = 2$。它由原序列的前两个元素 *{5, 2}* 排序后得到。
- **处理第三个元素 *(4)***: 将 *4* 插入到 *{2, 5}* 中。已排序子序列为 *{2, 4, 5}*。长度 $k = 3$。它由原序列的前三个元素 *{5, 2, 4}* 排序后得到。
- **处理第四个元素 *(1)***: 将 *1* 插入到 *{2, 4, 5}* 中。已排序子序列为 *{1, 2, 4, 5}*。长度 $k = 4$。它由原序列的前四个元素 *{5, 2, 4, 1}* 排序后得到。
- **处理第五个元素 *(3)***: 将 *3* 插入到 *{1, 2, 4, 5}* 中。已排序子序列为 *{1, 2, 3, 4, 5}*。长度 $k = 5$。它由原序列的前五个元素 *{5, 2, 4, 1, 3}* 排序后得到。

分析选项:

- ***A.* 其中的元素是整个序列中最小的 *k* 个元素**: 错误。例如，当 $k = 1$ 时，已排序子序列是 *{5}*，但 *5* 不是整个序列中最小的元素。
- ***B.* 其中的元素是整个序列中最大的 *k* 个元素**: 错误。
- ***C.* 其中的元素是原序列中位于前方的 *k* 个元素**: 正确。已排序子序列总是由原序列中从第 *0* 个到第 $k - 1$ 个元素（即前 k 个元素）经过排序后组成的。
- ***D.* 其中的元素是原序列中位于后方的 *k* 个元素**: 错误。

60

> **插入排序中的某一次插入后得到序列 {2, 7, 13, 5, 3, 19, 17}，此时已排序部分有 3 个元素。又经过 2 趟迭代后的结果是::**
> ① {2, 3, 5, 7, 13, 17, 19}
> ② {2, 3, 5, 7, 13, 19, 17}
> ③ {2, 5, 7, 3, 13, 19, 17}
> ④ {2, 3, 5, 13, 7, 17, 19}

Solution 20. 正确答案为 *B*。

当前序列 *V = {2, 7, 13, 5, 3, 19, 17}*。已排序部分有 *3* 个元素，即 *V[0...2] = {2, 7, 13}*。未排序部分的第一个元素是 *V[3] = 5*。

第 1 趟迭代 (插入元素 5):

- 取出元素 ‘*e* = *5*‘。
- 将 ‘*e*‘ 与已排序部分的 ‘*V[2]=13*‘ 比较: ‘*5* < *13*‘。‘*13*‘ 右移。*V* 变为 *{2, 7, _, 13, 3, 19, 17}* (概念上)。
- 将 ‘*e*‘ 与 ‘*V[1]=7*‘ 比较: ‘*5* < *7*‘。‘*7*‘ 右移。*V* 变为 *{2, _, 7, 13, 3, 19, 17}*。
- 将 ‘*e*‘ 与 ‘*V[0]=2*‘ 比较: ‘*5* > *2*‘。‘*5*‘ 插入到 ‘*2*‘ 之后。
- 序列变为 *V* = *{2, 5, 7, 13, 3, 19, 17}*。
- 已排序部分: *{2, 5, 7, 13}*。未排序部分: *{3, 19, 17}*。

第 2 趟迭代 (插入元素 3):

- 当前序列 *V* = *{2, 5, 7, 13, 3, 19, 17}*。
- 已排序部分 *V[0...3]* = *{2, 5, 7, 13}*。
- 未排序部分的第一个元素是 *V[4]* = *3*。
- 取出元素 ‘*e* = *3*‘。
- 将 ‘*e*‘ 与 ‘*V[3]=13*‘ 比较: ‘*3* < *13*‘。‘*13*‘ 右移。*V* 变为 *{2, 5, 7, _, 13, 19, 17}*。
- 将 ‘*e*‘ 与 ‘*V[2]=7*‘ 比较: ‘*3* < *7*‘。‘*7*‘ 右移。*V* 变为 *{2, 5, _, 7, 13, 19, 17}*。
- 将 ‘*e*‘ 与 ‘*V[1]=5*‘ 比较: ‘*3* < *5*‘。‘*5*‘ 右移。*V* 变为 *{2, _, 5, 7, 13, 19, 17}*。
- 将 ‘*e*‘ 与 ‘*V[0]=2*‘ 比较: ‘*3* > *2*‘。‘*3*‘ 插入到 ‘*2*‘ 之后。
- 序列变为 *V* = *{2, 3, 5, 7, 13, 19, 17}*。
- 已排序部分: *{2, 3, 5, 7, 13}*。未排序部分: *{19, 17}*。

经过 *2* 趟迭代后的结果是 *{2, 3, 5, 7, 13, 19, 17}*。

这与选项 *B* 相符。

61

> **一个序列的逆序数 T 定义为该序列中的逆序对总数，规模为 n 的列表中插入排序进行的元素比较总次数为: :**
>
> ① O(n+ T$\log_2$(T))
> ② O(n+T)
> ③ O(n^2 + $\log_2$(T))
> ④ O(T)

Solution 21. 正确答案为 *B*。

插入排序在将元素 ‘*A[j]*‘ 插入到已排序的子序列 ‘*A[0...j-1]*‘ 时，会将其与 ‘*A[j-1]*‘, ‘*A[j-2]*‘, ... 逐个比较，直到找到一个不大于 ‘*A[j]*‘ 的元素或者到达子序列的开头。每当 ‘*A[j]*‘ 小于它正在比较的元素 ‘*A[i]*‘ (其中 ‘*i* < *j*‘) 时，这意味着 ‘*(A[i], A[j])*‘ 形成了一个逆序对（在原始位置上，‘*A[i]*‘ 在 ‘*A[j]*‘ 之前，但 ‘*A[i]* > *A[j]*‘）。这个比较会导致 ‘*A[i]*‘ 向右移动，‘*A[j]*‘ 继续向前比较。

- 对于每一个逆序对 $(A[k], A[j])$ 其中 $k < j$ 且 $A[k] > A[j]$，当元素 $A[j]$ 被插入时，它必然会与 $A[k]$ 进行一次比较（并越过它）。因此，至少有 T 次比较是由于逆序对的存在而发生的。
- 此外，对于每个被插入的元素 $A[j]$ (从 $j = 1$ 到 $n-1$)，即使它大于所有在它之前的已排序元素（即没有与它形成逆序对的元素在它前面），它仍然需要进行一次比较来确定它的最终位置（即与它前面的那个元素比较，发现自己更大，然后停止）。这部分比较有 $n-1$ 次。

更准确地说，当第 j 个元素（从索引 *1* 到 $n-1$）被插入时，它会进行 $k_j + 1$ 次比较，其中 k_j 是在已排序部分中比它大的元素的数量（这些元素会被右移）。k_j 也正好是这个元素与前面元素形成的逆序对的数

量。总比较次数 = $\sum_{j=1}^{n-1}(k_j+1) = (\sum_{j=1}^{n-1} k_j) + \sum_{j=1}^{n-1} 1$. $\sum_{j=1}^{n-1} k_j$ 正是总的逆序对数量 T。$\sum_{j=1}^{n-1} 1 = n-1$. 所以，总比较次数是 $T+(n-1)$。

因此，时间复杂度是 $O(T+n-1) = O(n+T)$。

62

> 长度为 n 的列表，被等分为 n/k 段，每段长度为 k，不同段之间的元素不存在逆序。对该列表进行插入排序的最坏时间复杂度为：A. O(n^2) B. O(nk) C. O(n^2/k) D. O(n^{2}k)

Solution 22. 正确答案是 B。

详细解答:

设列表为 L，长度为 n。列表被等分为 $m = n/k$ 段，记为 $S_1, S_2, \ldots, S_m$。每段 S_i 的长度为 k。条件"不同段之间的元素不存在逆序"意味着：对于任意 $i < j$，以及任意元素 $x \in S_i$ 和任意元素 $y \in S_j$，都有 $x \leq y$。

我们来分析插入排序的过程：插入排序从列表的第二个元素开始，逐个将其插入到前面已经排好序的部分中。

考虑插入列表中的第 p 个元素 $L[p]$（从 *0* 开始计数或从 *1* 开始计数，分析过程类似，这里假设从 *1* 开始计数，则 $L[1\ldots p-1]$ 是已排序部分）。假设元素 $L[p]$ 属于段 S_j。

当我们将 $L[p]$ 向左移动以插入到正确位置时，它会与 $L[p-1], L[p-2], \ldots$ 等元素进行比较。

- ***(1)*** **与前面段的元素比较**：如果 $L[p]$ 与一个属于前面某个段 S_i *(*其中 $i < j$*)* 的元素 $L[q]$ *(*$q < p$*)* 进行比较，根据题目条件，$L[q] \leq L[p]$。这意味着 $L[p]$ 不需要移动到 $L[q]$ 的左边。因此，$L[p]$ 的插入过程不会跨越到它所在段 S_j 之前任何段的元素的左侧。
- ***(2)*** **与本段内的元素比较**：$L[p]$ 的比较和移动操作实际上只局限于其自身所在的段 S_j 内部，即与 S_j 中那些已经在 $L[p]$ 左边并且已经（相对）排序的元素进行比较。
- ***(3)*** **单个元素插入的代价**：对于段 S_j 中的任何一个元素，当它被插入时，它最多需要与该段内已经处理过的其它元素（最多 $k-1$ 个）进行比较和移动。在最坏情况下（例如，当前元素是其段内已处理元素中最小的），插入这个元素需要 $O(k)$ 次比较和 $O(k)$ 次移动。所以，插入一个元素到其所在段的正确位置，最坏时间复杂度为 $O(k)$。
- ***(4)*** **总时间复杂度**：列表中总共有 n 个元素。由于每个元素的插入操作，其比较和移动的范围被限制在其自身长度为 k 的段内，所以每个元素插入的最坏时间复杂度为 $O(k)$。因此，对整个列表进行插入排序的总的最坏时间复杂度为 $n \times O(k) = O(nk)$。

所以，最坏时间复杂度为 $O(nk)$。