选择排序（selection sort）的工作原理非常简单：开启一个循环，每轮从未排序区间选择最小的元素，将其放到已排序区间的末尾。

设数组的长度为 n ，选择排序的算法流程如图 11-2 所示。

1. 初始状态下，所有元素未排序，即未排序（索引）区间为 $[0, n-1]$ 。
2. 选取区间 $[0, n-1]$ 中的最小元素，将其与索引 0 处的元素交换。完成后，数组前 1 个元素已排序。
3. 选取区间 $[1, n-1]$ 中的最小元素，将其与索引 1 处的元素交换。完成后，数组前 2 个元素已排序。
4. 以此类推。经过 $n-1$ 轮选择与交换后，数组前 $n-1$ 个元素已排序。
5. 仅剩的一个元素必定是最大元素，无须排序，因此数组排序完成。

- **时间复杂度为 $O(n^2)$、非自适应排序**：外循环共 $n-1$ 轮，第一轮的未排序区间长度为 n ，最后一轮的未排序区间长度为 2 ，即各轮外循环分别包含 n、$n-1$、$\dots$、3、2 轮内循环，求和为 $\frac{(n-1)(n+2)}{2}$ 。
- **空间复杂度为 $O(1)$、原地排序**：指针 i 和 j 使用常数大小的额外空间。
- **非稳定排序**：如图 11-3 所示，元素 `nums[i]` 有可能被交换至与其相等的元素的右边，导致两者的相对顺序发生改变。

冒泡排序（bubble sort）通过连续地比较与交换相邻元素实现排序。这个过程就像气泡从底部升到顶部一样，因此得名冒泡排序。

如图 11-4 所示，冒泡过程可以利用元素交换操作来模拟：从数组最左端开始向右遍历，依次比较相邻元素大小，如果“左元素 > 右元素”就交换二者。遍历完成后，最大的元素会被移动到数组的最右端。

设数组的长度为 n ，冒泡排序的步骤如图 11-5 所示。

1. 首先，对 n 个元素执行“冒泡”，**将数组的最大元素交换至正确位置**。
2. 接下来，对剩余 $n-1$ 个元素执行“冒泡”，**将第二大元素交换至正确位置**。
3. 以此类推，经过 $n-1$ 轮“冒泡”后，**前 $n-1$ 大的元素都被交换至正确位置**。
4. 仅剩的一个元素必定是最小元素，无须排序，因此数组排序完成。

- **时间复杂度为 $O(n^2)$、自适应排序**：各轮“冒泡”遍历的数组长度依次为 $n-1$、$n-2$、$\dots$、2、1 ，总和为 $(n-1)n/2$ 。在引入 `flag` 优化后，最佳时间复杂度可达到 $O(n)$ 。
- **空间复杂度为 $O(1)$、原地排序**：指针 i 和 j 使用常数大小的额外空间。
- **稳定排序**：由于在“冒泡”中遇到相等元素不交换。

插入排序（insertion sort）是一种简单的排序算法，它的工作原理与手动整理一副牌的过程非常相似。

具体来说，我们在未排序区间选择一个基准元素，将该元素与其左侧已排序区间的元素逐一比较大小，并将该元素插入到正确的位置。

图 11-6 展示了数组插入元素的操作流程。设基准元素为 `base` ，我们需要将从目标索引到 `base` 之间的所有元素向右移动一位，然后将 `base` 赋值给目标索引。

插入排序的整体流程如图 11-7 所示。

1. 初始状态下，数组的第 1 个元素已完成排序。
2. 选取数组的第 2 个元素作为 `base` ，将其插入到正确位置后，**数组的前 2 个元素已排序**。
3. 选取第 3 个元素作为 `base` ，将其插入到正确位置后，**数组的前 3 个元素已排序**。
4. 以此类推，在最后一轮中，选取最后一个元素作为 `base` ，将其插入到正确位置后，**所有元素均已排序**。

- **时间复杂度为 $O(n^2)$、自适应排序**：在最差情况下，每次插入操作分别需要循环 $n-1$、$n-2$、$\dots$、2、1 次，求和得到 $(n-1)n/2$ ，因此时间复杂度为 $O(n^2)$ 。在遇到有序数据时，插入操作会提前终止。当输入数组完全有序时，插入排序达到最佳时间复杂度 $O(n)$ 。
- **空间复杂度为 $O(1)$、原地排序**：指针 i 和 j 使用常数大小的额外空间。
- **稳定排序**：在插入操作过程中，我们会将元素插入到相等元素的右侧，不会改变它们的顺序。

插入排序的时间复杂度为 $O(n^2)$ ，而我们即将学习的快速排序的时间复杂度为 $O(n \log n)$ 。尽管插入排序的时间复杂度更高，**但在数据量较小的情况下，插入排序通常更快**。

这个结论与线性查找和二分查找的适用情况的结论类似。快速排序这类 $O(n \log n)$ 的算法属于基于分治策略的排序算法，往往包含更多单元计算操作。而在数据量较小时，n^2 和 $n \log n$ 的数值比较接近，复杂度不占主导地位，每轮中的单元操作数量起到决定性作用。

实际上，许多编程语言（例如 Java）的内置排序函数采用了插入排序，大致思路为：对于长数组，采用基于分治策略的排序算法，例如快速排序；对于短数组，直接使用插入排序。

虽然冒泡排序、选择排序和插入排序的时间复杂度都为 $O(n^2)$ ，但在实际情况中，**插入排序的使用频率显著高于冒泡排序和选择排序**，主要有以下原因。

- 冒泡排序基于元素交换实现，需要借助一个临时变量，共涉及 3 个单元操作；插入排序基于元素赋值实现，仅需 1 个单元操作。因此，**冒泡排序的计算开销通常比插入排序更高**。
- 选择排序在任何情况下的时间复杂度都为 $O(n^2)$ 。**如果给定一组部分有序的数据，插入排序通常比选择排序效率更高**。
- 选择排序不稳定，无法应用于多级排序。

快速排序（quick sort）是一种基于分治策略的排序算法，运行高效，应用广泛。

快速排序的核心操作是“哨兵划分”，其目标是：选择数组中的某个元素作为“基准数”，将所有小于基准数的元素移到其左侧，而大于基准数的元素移到其右侧。具体来说，哨兵划分的流程如图 11-8 所示。

1. 选取数组最左端元素作为基准数，初始化两个指针 `i` 和 `j` 分别指向数组的两端。
2. 设置一个循环，在每轮中使用 `i` （ `j` ）分别寻找第一个比基准数大（小）的元素，然后交换这两个元素。
3. 循环执行步骤 `2.` ，直到 `i` 和 `j` 相遇时停止，最后将基准数交换至两个子数组的分界线。

快速排序的整体流程如图 11-9 所示。

1. 首先，对原数组执行一次“哨兵划分”，得到未排序的左子数组和右子数组。
2. 然后，对左子数组和右子数组分别递归执行“哨兵划分”。
3. 持续递归，直至子数组长度为 1 时终止，从而完成整个数组的排序。

从名称上就能看出，快速排序在效率方面应该具有一定的优势。尽管快速排序的平均时间复杂度与“归并排序”和“堆排序”相同，但通常快速排序的效率更高，主要有以下原因。

- **出现最差情况的概率很低**：虽然快速排序的最差时间复杂度为 $O(n^2)$ ，没有归并排序稳定，但在绝大多数情况下，快速排序能在 $O(n \log n)$ 的时间复杂度下运行。
- **缓存使用效率高**：在执行哨兵划分操作时，系统可将整个子数组加载到缓存，因此访问元素的效率较高。而像“堆排序”这类算法需要跳跃式访问元素，从而缺乏这一特性。
- **复杂度的常数系数小**：在上述三种算法中，快速排序的比较、赋值、交换等操作的总数量最少。这与“插入排序”比“冒泡排序”更快的原因类似。

- **时间复杂度为 $O(n \log n)$、自适应排序**：在平均情况下，哨兵划分的递归层数为 $\log n$ ，每层中的总循环数为 n ，总体使用 $O(n \log n)$ 时间。在最差情况下，每轮哨兵划分操作都将长度为 n 的数组划分为长度为 0 和 $n - 1$ 的两个子数组，此时递归层数达到 n ，每层中的循环数为 n ，总体使用 $O(n^2)$ 时间。
- **空间复杂度为 $O(n)$、原地排序**：在输入数组完全倒序的情况下，达到最差递归深度 n ，使用 $O(n)$ 栈帧空间。排序操作是在原数组上进行的，未借助额外数组。
- **非稳定排序**：在哨兵划分的最后一步，基准数可能会被交换至相等元素的右侧。

快速排序在某些输入下的时间效率可能降低。举一个极端例子，假设输入数组是完全倒序的，由于我们选择最左端元素作为基准数，那么在哨兵划分完成后，基准数被交换至数组最右端，导致左子数组长度为 $n - 1$、右子数组长度为 0 。如此递归下去，每轮哨兵划分后都有一个子数组的长度为 0 ，分治策略失效，快速排序退化为“冒泡排序”的近似形式。

为了尽量避免这种情况发生，**我们可以优化哨兵划分中的基准数的选取策略**。例如，我们可以随机选取一个元素作为基准数。然而，如果运气不佳，每次都选到不理想的基准数，效率仍然不尽如人意。

需要注意的是，编程语言通常生成的是“伪随机数”。如果我们针对伪随机数序列构建一个特定的测试样例，那么快速排序的效率仍然可能劣化。

为了进一步改进，我们可以在数组中选取三个候选元素（通常为数组的首、尾、中点元素），**并将这三个候选元素的中位数作为基准数**。这样一来，基准数“既不太小也不太大”的概率将大幅提升。当然，我们还可以选取更多候选元素，以进一步提高算法的稳健性。采用这种方法后，时间复杂度劣化至 $O(n^2)$ 的概率大大降低。

在某些输入下，快速排序可能占用空间较多。以完全有序的输入数组为例，设递归中的子数组长度为 m ，每轮哨兵划分操作都将产生长度为 0 的左子数组和长度为 $m - 1$ 的右子数组，这意味着每一层递归调用减少的问题规模非常小（只减少一个元素），递归树的高度会达到 $n - 1$ ，此时需要占用 $O(n)$ 大小的栈帧空间。

为了防止栈帧空间的累积，我们可以在每轮哨兵排序完成后，比较两个子数组的长度，**仅对较短的子数组进行递归**。由于较短子数组的长度不会超过 $n/2$ ，因此这种方法能确保递归深度不超过 $\log n$ ，从而将最差空间复杂度优化至 $O(\log n)$ 。代码如下所示：

归并排序（merge sort）是一种基于分治策略的排序算法，包含图 11-10 所示的“划分”和“合并”阶段。

1. **划分阶段**：通过递归不断地将数组从中点处分开，将长数组的排序问题转换为短数组的排序问题。
2. **合并阶段**：当子数组长度为 1 时终止划分，开始合并，持续地将左右两个较短的有序数组合并为一个较长的有序数组，直至结束。

如图 11-11 所示，“划分阶段”从顶至底递归地将数组从中点切分为两个子数组。

1. 计算数组中点 `mid` ，递归划分左子数组（区间 `[left, mid]` ）和右子数组（区间 `[mid + 1, right]` ）。
2. 递归执行步骤 `1.` ，直至子数组区间长度为 1 时终止。

“合并阶段”从底至顶地将左子数组和右子数组合并为一个有序数组。需要注意的是，从长度为 1 的子数组开始合并，合并阶段中的每个子数组都是有序的。

观察发现，归并排序与二叉树后序遍历的递归顺序是一致的。

- **后序遍历**：先递归左子树，再递归右子树，最后处理根节点。
- **归并排序**：先递归左子数组，再递归右子数组，最后处理合并。

归并排序的实现如以下代码所示。请注意， `nums` 的待合并区间为 `[left, right]` ，而 `tmp` 的对应区间为 `[0, right - left]` 。

- **时间复杂度为 $O(n \log n)$、非自适应排序**：划分产生高度为 $\log n$ 的递归树，每层合并的总操作数量为 n ，因此总体时间复杂度为 $O(n \log n)$ 。
- **空间复杂度为 $O(n)$、非原地排序**：递归深度为 $\log n$ ，使用 $O(\log n)$ 大小的栈帧空间。合并操作需要借助辅助数组实现，使用 $O(n)$ 大小的额外空间。
- **稳定排序**：在合并过程中，相等元素的次序保持不变。

11.6.3 链表排序

对于链表，归并排序相较于其他排序算法具有显著优势，**可以将链表排序任务的空间复杂度优化至 $O(1)$**

- **划分阶段**：可以使用“迭代”替代“递归”来实现链表划分工作，从而省去递归使用的栈帧空间。
- **合并阶段**：在链表中，节点增删操作仅需改变引用（指针）即可实现，因此合并阶段（将两个短有序链表合并为一个长有序链表）无须创建额外链表。

具体实现细节比较复杂，有兴趣的读者可以查阅相关资料进行学习。

堆排序（heap sort）是一种基于堆数据结构实现的高效排序算法。我们可以利用已经学过的“建堆操作”和“元素出堆操作”实现堆排序。

1. 输入数组并建立小顶堆，此时最小元素位于堆顶。
2. 不断执行出堆操作，依次记录出堆元素，即可得到从小到大排序的序列。

以上方法虽然可行，但需要借助一个额外数组来保存弹出的元素，比较浪费空间。在实际中，我们通常使用一种更加优雅的实现方式。

11.7.1 算法流程

设数组的长度为 n ，堆排序的流程如图 11-12 所示。

1. 输入数组并建立大顶堆。完成后，最大元素位于堆顶。
2. 将堆顶元素（第一个元素）与堆底元素（最后一个元素）交换。完成交换后，堆的长度减 1 ，已排序元素数量加 1 。
3. 从堆顶元素开始，从顶到底执行堆化操作（sift down）。完成堆化后，堆的性质得到修复。
4. 循环执行第 `2.` 步和第 `3.` 步。循环 $n - 1$ 轮后，即可完成数组排序。

- **时间复杂度为 $O(n \log n)$、非自适应排序**：建堆操作使用 $O(n)$ 时间。从堆中提取最大元素的时间复杂度为 $O(\log n)$ ，共循环 $n - 1$ 轮。
- **空间复杂度为 $O(1)$、原地排序**：几个指针变量使用 $O(1)$ 空间。元素交换和堆化操作都是在原数组上进行的。
- **非稳定排序**：在交换堆顶元素和堆底元素时，相等元素的相对位置可能发生变化。

- **时间复杂度为 $O(n \log n)$、非自适应排序**：建堆操作使用 $O(n)$ 时间。从堆中提取最大元素的时间复杂度为 $O(\log n)$ ，共循环 $n - 1$ 轮。
- **空间复杂度为 $O(1)$、原地排序**：几个指针变量使用 $O(1)$ 空间。元素交换和堆化操作都是在原数组上进行的。
- **非稳定排序**：在交换堆顶元素和堆底元素时，相等元素的相对位置可能发生变化。

前述几种排序算法都属于“基于比较的排序算法”，它们通过比较元素间的大小来实现排序。此类排序算法的时间复杂度无法超越 $O(n \log n)$ 。接下来，我们将探讨几种“非比较排序算法”，它们的时间复杂度可以达到线性阶。

桶排序（bucket sort）是分治策略的一个典型应用。它通过设置一些具有大小顺序的桶，每个桶对应一个数据范围，将数据平均分配到各个桶中；然后，在每个桶内部分别执行排序；最终按照桶的顺序将所有数据合并。

11.8.1 算法流程

考虑一个长度为 n 的数组，其元素是范围 $[0, 1)$ 内的浮点数。桶排序的流程如图 11-13 所示。

1. 初始化 k 个桶，将 n 个元素分配到 k 个桶中。
2. 对每个桶分别执行排序（这里采用编程语言的内置排序函数）。
3. 按照桶从小到大的顺序合并结果。

桶排序适用于处理体量很大的数据。例如，输入数据包含 100 万个元素，由于空间限制，系统内存无法一次性加载所有数据。此时，可以将数据分成 1000 个桶，然后分别对每个桶进行排序，最后将结果合并。

- **时间复杂度为 $O(n + k)$**：假设元素在各个桶内平均分布，那么每个桶内的元素数量为 $\frac{n}{k}$ 。假设排序单个桶使用 $O(\frac{n}{k} \log \frac{n}{k})$ 时间，则排序所有桶使用 $O(n \log \frac{n}{k})$ 时间。**当桶数量 k 比较大时，时间复杂度则趋向于 $O(n)$** 。合并结果时需要遍历所有桶和元素，花费 $O(n + k)$ 时间。
- **自适应排序**：在最差情况下，所有数据被分配到一个桶中，且排序该桶使用 $O(n^2)$ 时间。
- **空间复杂度为 $O(n + k)$、非原地排序**：需要借助 k 个桶和总共 n 个元素的额外空间。
- 桶排序是否稳定取决于排序桶内元素的算法是否稳定。

11.8.3 如何实现平均分配

桶排序的时间复杂度理论上可以达到 $O(n)$ ，**关键在于将元素均匀分配到各个桶中**，因为实际数据往往不是均匀分布的。例如，我们想要将淘宝上的所有商品按价格范围平均分配到 10 个桶中，但商品价格分布不均，低于 100 元的非常多，高于 1000 元的非常少。若将价格区间平均划分为 10 个，各个桶中的商品数量差距会非常大。

为实现平均分配，我们可以先设定一条大致的分界线，将数据粗略地分到 3 个桶中。**分配完毕后，再将商品较多的桶继续划分为 3 个桶，直至所有桶中的元素数量大致相等**。

如图 11-14 所示，这种方法本质上是创建一棵递归树，目标是让叶节点的值尽可能平均。当然，不一定要每轮将数据划分为 3 个桶，具体划分方式可根据数据特点灵活选择。

计数排序（counting sort）通过统计元素数量来实现排序，通常应用于整数数组。

11.9.1 简单实现

先来看一个简单的例子。给定一个长度为 n 的数组 `nums` ，其中的元素都是“非负整数”，计数排序的整体流程如图 11-16 所示。

1. 遍历数组，找出其中的最大数字，记为 m ，然后创建一个长度为 $m + 1$ 的辅助数组 `counter` 。
2. **借助 `counter` 统计 `nums` 中各数字的出现次数**，其中 `counter[num]` 对应数字 `num` 的出现次数。统计方法很简单，只需遍历 `nums` （设当前数字为 `num` ），每轮将 `counter[num]` 增加 1 即可。
3. **由于 `counter` 的各个索引天然有序，因此相当于所有数字已经排序好了**。接下来，我们遍历 `counter` ，根据各数字出现次数从小到大的顺序填入 `nums` 即可。

细心的读者可能发现了，**如果输入数据是对象，上述步骤 `3.` 就失效了**。假设输入数据是商品对象，我们想按照商品价格（类的成员变量）对商品进行排序，而上述算法只能给出价格的排序结果。

那么如何才能得到原数据的排序结果呢？我们首先计算 `counter` 的“前缀和”。顾名思义，索引 `i` 处的前缀和 `prefix[i]` 等于数组前 `i` 个元素之和：

$$\text{prefix}[i] = \sum_{j=0}^{i} \text{counter}[\text{j}]$$

前缀和具有明确的意义，`prefix[num] - 1` 代表元素 `num` 在结果数组 `res` 中最后一次出现的索引。这个信息非常关键，因为它告诉我们各个元素应该出现在结果数组的哪个位置。接下来，我们倒序遍历原数组 `nums` 的每个元素 `num` ，在每轮迭代中执行以下两步。

1. 将 `num` 填入数组 `res` 的索引 `prefix[num] - 1` 处。
2. 令前缀和 `prefix[num]` 减小 1 ，从而得到下次放置 `num` 的索引。

遍历完成后，数组 `res` 中就是排序好的结果，最后使用 `res` 覆盖原数组 `nums` 即可。图 11-17 展示了完整的计数排序流程。

- **时间复杂度为 $O(n + m)$、非自适应排序**：涉及遍历 `nums` 和遍历 `counter` ，都使用线性时间。一般情况下 $n \gg m$ ，时间复杂度趋于 $O(n)$ 。
- **空间复杂度为 $O(n + m)$、非原地排序**：借助了长度分别为 n 和 m 的数组 `res` 和 `counter` 。
- **稳定排序**：由于向 `res` 中填充元素的顺序是“从右向左”的，因此倒序遍历 `nums` 可以避免改变相等元素之间的相对位置，从而实现稳定排序。实际上，正序遍历 `nums` 也可以得到正确的排序结果，但结果是非稳定的。

11.9.4 局限性

看到这里，你也许会觉得计数排序非常巧妙，仅通过统计数量就可以实现高效的排序。然而，使用计数排序的前置条件相对较为严格。

计数排序只适用于非负整数。若想将其用于其他类型的数据，需要确保这些数据可以转换为非负整数，并且在转换过程中不能改变各个元素之间的相对大小关系。例如，对于包含负数的整数数组，可以先给所有数字加上一个常数，将全部数字转化为正数，排序完成后再转换回去。

计数排序适用于数据量大但数据范围较小的情况。比如，在上述示例中 m 不能太大，否则会占用过多空间。而当 $n \ll m$ 时，计数排序使用 $O(m)$ 时间，可能比 $O(n \log n)$ 的排序算法还要慢。

上一节介绍了计数排序，它适用于数据量 n 较大但数据范围 m 较小的情况。假设我们需要对 $n = 10^6$ 个学号进行排序，而学号是一个 8 位数字，这意味着数据范围 $m = 10^8$ 非常大，使用计数排序需要分配大量内存空间，而基数排序可以避免这种情况。

基数排序（radix sort）的核心思想与计数排序一致，也通过统计个数来实现排序。在此基础上，基数排序利用数字各位之间的递进关系，依次对每一位进行排序，从而得到最终的排序结果。

11.10.1 算法流程

以学号数据为例，假设数字的最低位是第 1 位，最高位是第 8 位，基数排序的流程如图 11-18 所示。

1. 初始化位数 $k = 1$。
2. 对学号的第 k 位执行“计数排序”。完成后，数据会根据第 k 位从小到大排序。
3. 将 k 增加 1，然后返回步骤 `2.` 继续迭代，直到所有位都排序完成后结束。

相较于计数排序，基数排序适用于数值范围较大的情况，**但前提是数据必须可以表示为固定位数的格式，且位数不能过大**。例如，浮点数不适合使用基数排序，因为其位数 k 过大，可能导致时间复杂度 $O(nk) \gg O(n^2)$。

- **时间复杂度为 $O(nk)$、非自适应排序**：设数据量为 n、数据为 d 进制、最大位数为 k，则对某一位执行计数排序使用 $O(n + d)$ 时间，排序所有 k 位使用 $O((n + d)k)$ 时间。通常情况下，d 和 k 都相对较小，时间复杂度趋向 $O(n)$。
- **空间复杂度为 $O(n + d)$、非原地排序**：与计数排序相同，基数排序需要借助长度为 n 和 d 的数组 `res` 和 `counter`。
- **稳定排序**：当计数排序稳定时，基数排序也稳定；当计数排序不稳定时，基数排序无法保证得到正确的排序结果。

		时间复杂度			空间复杂度	稳定性	就地性	自适应性	基于比较
		最佳	平均	最差	最差				
遍历排序 $O(n^2)$	选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	非稳定	原地	非自适应	比较
	冒泡排序	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	稳定	原地	自适应	比较
	插入排序	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	稳定	原地	自适应	比较
分治排序 $O(n \log n)$	快速排序	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	非稳定	原地	自适应	比较
	归并排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	稳定	非原地	非自适应	比较
	堆排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	非稳定	原地	非自适应	比较
线性排序 $O(n)$	桶排序	$O(n + k)$	$O(n + k)$	$O(n^2)$	$O(n + k)$	稳定	非原地	自适应	非比较
	计数排序	$O(n + m)$	$O(n + m)$	$O(n + m)$	$O(n + m)$	稳定	非原地	非自适应	非比较
	基数排序	$O(n\ k)$	$O(n\ k)$	$O(n\ k)$	$O(n + b)$	稳定	非原地	非自适应	非比较

差 中 优

n 为数据量大小
桶排序中，**k** 为桶数量
计数排序中，**m** 为数据范围
基数排序中，**k** 为最大位数，数据为 **b** 进制

	迭代	递归
实现方式	循环结构	函数调用自身
时间效率	效率通常较高，无函数调用开销	每次函数调用都会产生开销
内存使用	通常使用固定大小的内存空间	累积函数调用可能使用大量的栈帧空间
适用问题	适用于简单循环任务，代码直观、可读性好	适用于子问题分解，如树、图、分治、回溯等，代码结构简洁、清晰

	数组	链表
存储方式	连续内存空间	分散内存空间
容量扩展	长度不可变	可灵活扩展
内存效率	元素占用内存少、但可能浪费空间	元素占用内存多
访问元素	$O(1)$	$O(n)$
添加元素	$O(n)$	$O(1)$
删除元素	$O(n)$	$O(1)$