Big-O 符号（Ο） ——Big-O 符号专门描述最坏的情况。

Omega 符号 (Ω) – Omega(Ω) 符号具体描述了最佳情况。

西塔符号 (θ) – 该符号表示算法的平均复杂度。

两种基本类型的索引：

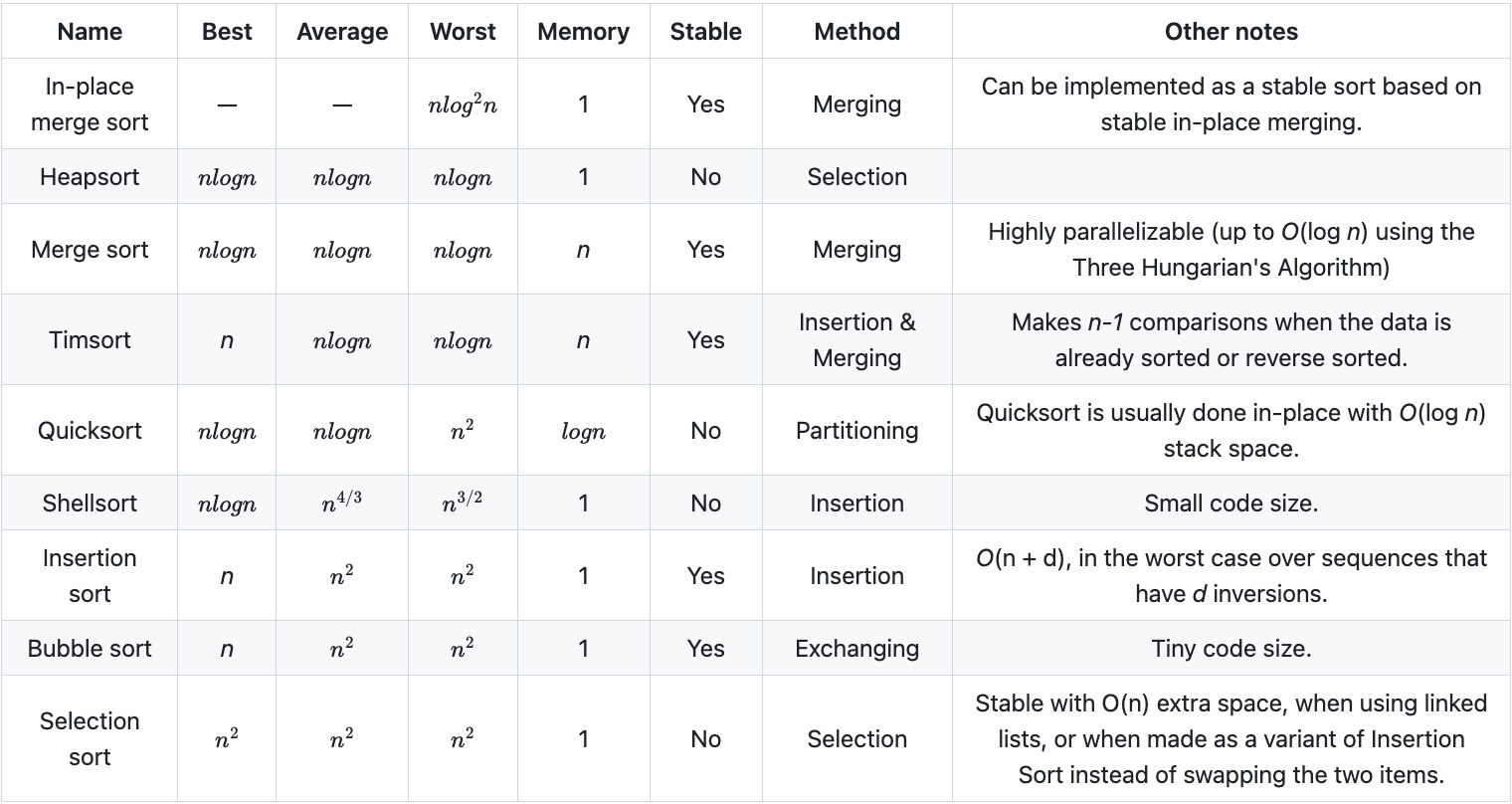
- 顺序索引：基于值的顺序排序

- 散列索引：基于将值平均分道若干散列桶中。一个值所属的散列桶是由散列函数（hash function）决定的。

数据的最小单位：数据项

数据的基本单位：数据元素

数据结构：带有结构的各数据元素的集合



排序算法有很多种类型。一些广泛使用的算法是：

1. 冒泡排序：冒泡排序是最简单的排序算法，其工作原理是如果相邻元素的顺序错误，则反复交换它们。该算法不适用于大型数据集，因为其平均和最坏情况的时间复杂度相当高。在冒泡排序算法中，从左侧遍历并比较相邻元素，较大的元素放在右侧。

这样，最大元素首先被移动到最右端。

然后继续此过程，找到第二大值并将其放置，依此类推，直到数据排序完毕。

时间复杂度：O(N2) 辅助空间：O(1)

冒泡排序的优点：容易理解和实现；不需要任何额外的内存空间；是一种稳定的排序算法，这意 味着具有相同键值的元素在排序输出中保持其相对顺序。

缺点：时间复杂度为 O(N2)，使其对于大型数据集的处理速度非常慢；限制算法的效率。

Q1. 冒泡排序的边界情况是什么？

当元素已经排序时，冒泡排序所需的时间最短（n 阶）。因此，最好事先检查数组是否已经排序，以避免 O(N2) 时间复杂度。冒泡排序中排序是就地进行的

1. 选择排序：选择排序是一种简单而有效的排序算法，其工作方式是反复从列表的未排序部分中选择最小（或最大）元素并将其移动到列表的已排序部分。

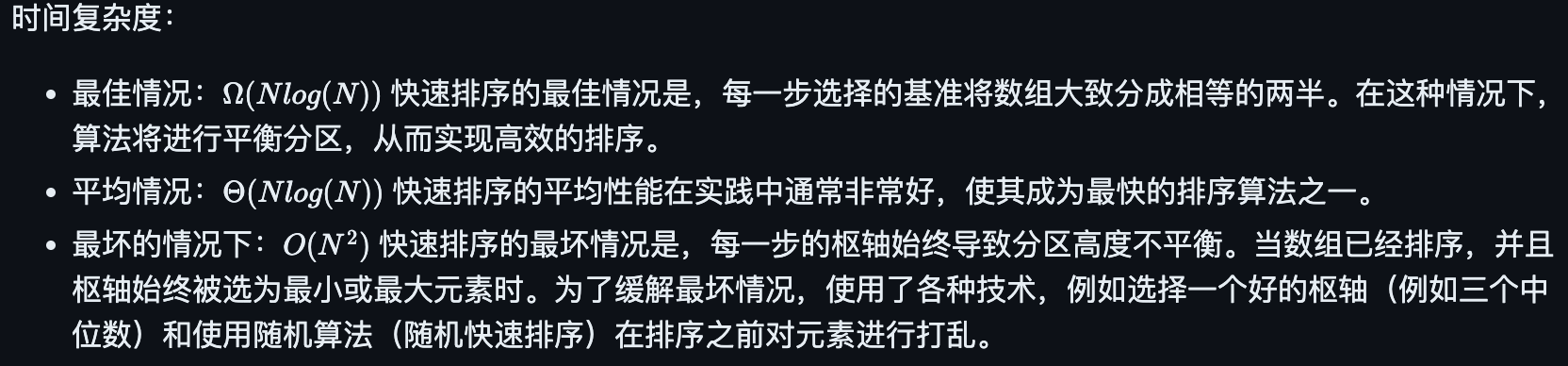
选择排序改进了冒泡排序，每次遍历列表时只进行一次交换。为了做到这一点，选择排序在遍历时寻找最大值，并在完成遍历后将其放置在正确的位置。与冒泡排序一样，在第一次遍历后，最大的项位于正确的位置。在第二次遍历后，次大项就位。这个过程继续进行，需要 n-1 次遍历才能对n 个项进行排序，因为最后一个项必须在第 (n-1) 次遍历后就位。

时间复杂度：选择排序的时间复杂度为O(N2)，因为有两个嵌套循环；辅助空间： O(1)

选择排序算法的优点：簡單易懂；适用于小型数据集。

选择排序算法的缺点：选择排序在最坏和平均情况下的时间复杂度都是 O(n^2)；在大型数据集上效果不佳；不保留具有相同键的项目的相对顺序，这意味着它不稳定。

1. 插入排序：将数组分为2端，有序数组和无序数组，依次将无序数组中的值插入到无序数组中。INSERTIONSORT 对两个大小相同的输入序列进行排序所花费的时间可能不同，具体取决于它们已经排序的程度。一般来说，算法所花费的时间会随着输入的大小而增加，因此传统上将程序的运行时间描述为其输入大小的函数。（最佳情况（输入数组已经排序）和最坏情况（输入数组反向排序））
2. 快速排序：quickSort 中的关键过程是分区（partition()）。分区的目的是将枢轴（任何元素都可以选择为枢轴）放置在排序数组中的正确位置，并将所有较小的元素放在枢轴的左边，将所有较大的元素放在枢轴的右边。将主元放置在正确位置后，对主元的每一侧递归进行分区，最终对数组进行排序。选择枢轴值有多种不同的方法。具体来说，我们可以尝试使用一种称为三中位数的技术来减轻不均匀划分的可能性。



辅助空间：如果不考虑递归堆栈空间，则为 O(1)。如果考虑递归堆栈空间，则在最坏情况下，快速排序的复杂度为 O(N)。

快速排序的优点：可以使解决问题变得更容易；对大型数据集非常有效；只需要少量的内存即可运行。

快速排序的缺点：当枢轴选择不当时时间复杂度为ON平方；对于小数据集来说快排不稳定

1. 归并排序：算法将数组分成两半，然后递归地对它们进行排序，最后将排好序的两半合并起来，是唯一保证 O(n log n) 的排序，即使在最坏的情况下也是如此。代价是归并排序会占用更多内存。归并排序的时间复杂度在所有 3 种情况下（最差、平均和最佳）均为 θ(Nlog(N))，辅助空间：O(N)。

归并排序的优点：稳定性；在大型数据集上也能表现良好；可并行化；通常用于外部排序。

归并排序的缺点：在排序过程中需要额外的内存来存储合并的子数组；非就地，这对于需要关注内存使用情况的应用程序来说可能是一个缺点；对于小型数据集而言并非总是最佳选择。

1. 希尔排序：主要是插入排序的一种变体。在插入排序中，我们只将元素向前移动一个位置。当一个元素必须向前移动很远时，就会涉及许多移动。希尔排序的思想是允许交换远处的项目。在希尔排序中，我们让数组 h 排序，以获得较大的 h 值。我们不断减少 h 的值，直到它变为 1。如果每个第 h 个元素的所有子列表都已排序，则称该数组是 h 排序的。

希尔排序应用：插入排序的替代，它需要很长时间才能完成给定的任务。

为了调用堆栈开销，我们使用希尔排序；当递归超过特定限制时，我们使用希尔排序。适用于中型到大型数据集，以插入排序来减少操作次数。



不是一种稳定排序，它是一种自适应排序算法

例题：

1. 用Huffman算法构造一个最优化二叉编码树：为了哈夫曼树，我们遵循一个重复的选择，每次选择两个最小的权值创建一个新的节点，直到只剩下一个节点为止。再计算哈夫曼树的带权外部路径长度（WPL）。
2. 用相邻连接矩阵法存储一个图时，在不考虑压缩存储的情况下，所占用的占用大小只与图中结束点个数有关，而与图的边数无关。
3. 问题适合应用动态规划算法的首选条件是它具有最优化子结构性质和重叠性质的子问题。

数据结构：

·栈：后进先出（堆盘子）：允许表达式内多种括号混合嵌套，检查表达式中括号是否正确配对的算法

·队列：先进先出（后进的在队尾）：

入队：将项目添加到队列。如果队列已满，则称为溢出条件 - 时间复杂度：O（1）

出队：从队列中移除一个项目。项目弹出的顺序与推送的顺序相同。如果队列为空，则称为下溢条件 - 时间复杂度：O(1)

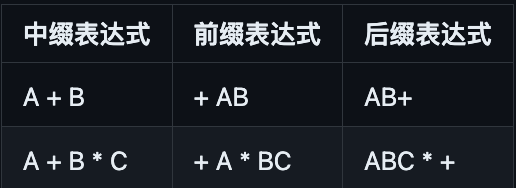
前端：从队列中获取前端项目 - 时间复杂度：O（1）

后方：从队列中获取最后一个项目 - 时间复杂度：O（1）

·线性表定义了元素按线性顺序排列的规则。 常见的线性表存储方法有阵列和链表【线性表的顺序存储与链式存储是两种常见存储形式；当表元素有序排序进行二分检索时，应采用（顺序存储）存储形式。】

·数组是一种存储系统，它按照指定的格式存储多个元素，每个元素的存储空间大小为O(1)，平均存储容量为O(n)

·链表是一种线性表，包含一个特定的对象包含的数据类型和操作数，链表的引用类型和操作数都是O(1)，也就是说，每个对象包含的数据类型和操作数都是O(n)。插入和删除不需要移动元素；不必事先估计存储空间；所需空间与线性表长度成正比；但不可随机访问任意元素



Q: 什么是 Python 的特殊方法（Magic Method） ? 请简述\_\_str\_\_, \_\_iter\_\_， \_\_contains\_\_三种特 殊方法的作用？

Python的特殊方法（也称为魔术方法或魔法方法）是一组在类中使用双下划线（\_\_）包围的预定义方法。这些方法提供了一种自定义类行为的机制，用于重载运算符、实现迭代、访问属性等。

\_\_str\_\_(self)： \_\_str\_\_方法用于返回对象的字符串表示形式。当我们使用print函数或者在字符串中引用对象时，会自动调用该方法。它通常被用于提供有意义的对象描述。例如，当我们定义一个名为Person的类并实现\_\_str\_\_方法时，可以返回该对象的姓名和年龄的字符串表示。

\_\_iter\_\_(self)： \_\_iter\_\_方法用于返回一个迭代器对象，使得我们可以在自定义的类上实现迭代功能。迭代器对象提供了\_\_next\_\_方法，用于按照定义的规则返回下一个值。通过实现\_\_iter\_\_方法，我们可以使用for循环等迭代器相关操作来遍历自定义类的实例。

\_\_contains\_\_(self, item)： \_\_contains\_\_方法用于确定对象是否包含特定的元素。当我们使用in运算符来检查一个对象是否包含某个元素时，会调用该方法。我们可以在自定义的类中实现\_\_contains\_\_方法来定义对象的包含逻辑。例如，当我们定义一个名为MyList的类，并实现\_\_contains\_\_方法时，可以检查列表中是否包含特定元素。

·树形数据结构的应用

文件系统；数据压缩：霍夫曼编码是一种流行的数据压缩技术，它涉及构建二叉树，其中叶子代表字符及其出现频率；编译器设计；数据库索引

·树的优点：树提供高效的搜索，对于 AVL 等平衡树，平均搜索时间为 O（log n）；使得组织和浏览大量信息变得容易；树的递归性质使其易于通过递归算法进行遍历和操作。

·树的缺点：不平衡树，意味着树的高度向一侧倾斜，这会导致搜索时间低效；树需要更多的内存空间，尤其是当树非常大时。

·n阶多叉树的缺点是：并非所有情况下都需要所有指针，因此存在大量内存浪费；子节点数量未知简单方法：为了将子节点的地址存储在节点中，我们可以使用数组或链表。

在链表中，我们无法随机访问任何子节点的地址。因此，这会很昂贵。

在数组中，我们可以随机访问任何子项的地址，但我们只能在其中存储固定数量的子项地址。

更好的方法：我们可以使用动态数组来存储子代的地址。我们可以随机访问任何子代的地址，并且向量的大小也不固定。

·前序遍历：先访问根节点，然后递归地前序遍历左子树，最后递归地前序遍历右子树。

中序遍历：先递归地中序遍历左子树，然后访问根节点，最后递归地中序遍历右子树。

后序遍历：先递归地后序遍历左子树，然后递归地后序遍历右子树，最后访问根节点。

·二叉堆的实现

结构性：为了使二叉堆能高效地工作，我们利用树的对数性质来表示它。为了保证对数性能，必须维持树的平衡。平衡的二叉树是指，其根节点的左右子树含有数量大致相等的节点。在实现二叉堆时，我们通过创建一棵完全二叉树complete binary tree来维持树的平衡。在完全二叉树中，除了最底层，其他每一层的节点都是满的。在最底层，我们从左往右填充节点。

·堆的有序性是指：对于堆中任意元素 x 及其父元素 p， p 都不大于 x

·二叉搜索树是一种有序的二叉树，它满足以下性质：

左子树中的所有节点的值都小于根节点的值。

右子树中的所有节点的值都大于根节点的值。

左子树和右子树也分别是二叉搜索树。

二叉搜索树实现快排这种方法的时间复杂度为 O(n log n)，其中 n 是数组的长度。每次插入操作都需要 O(log n) 的时间复杂度，总共进行 n-1 次插入操作。

·AVL 平衡树通过在每个节点上维护一个平衡因子（balance factor）来实现平衡。平衡因子是指节点的左子树高度与右子树高度之差的绝对值。通过不断调整树的结构，AVL 树能够保持树的平衡，使得在最坏情况下的查找、插入和删除操作的时间复杂度保持在 O(log n)。特点是在每次插入或删除节点时，会通过旋转操作来调整树的结构，使得平衡因子在特定的范围内，通常是 -1、0、1。这样的平衡状态能够保证树的高度始终保持在较小的范围内，提供了较快的查找和更新操作。如果平衡因子大于零，我们称之为左倾；如果平衡因子小于零，就是右倾；如果平衡因子等于零，那么树就是完全平衡的。插入新键：因为新叶子节点的平衡因子是零，所以新插节点没有什么限制条件。但插入新节点后，必须更新父节点的平衡因子。新的叶子节点对其父节点平衡因子的影响取决于它是左子节点还是右子节点。如果是右子节点，父节点的平衡因子减一。如果是左子节点，则父节点的平衡因子加一。

·并查集：在并查集中，每个元素都属于一个集合，并且这些集合之间是不相交的。为了高效地实现并查集操作，通常会使用树形结构来表示集合之间的关系。每个集合可以用一个树表示，其中树的根节点是集合的代表元素。使用邻接表来表示这种树形结构是一种常见的做法，其中每个节点存储其父节点的指针。

·前缀树：前缀树是一种用于存储字符串集合的数据结构，通常用于快速地进行字符串匹配和搜索。在前缀树中，每个节点代表一个字符，从根节点到叶子节点的路径表示一个字符串。为了表示字符串的结构，通常会使用邻接表来表示前缀树，其中每个节点存储一个字符以及指向子节点的指针列表。

·使用邻接表表示并查集和前缀树可以高效地支持各种操作，如并查集的合并和查找操作，以及前缀树的插入、搜索和删除操作。同时，邻接表也能够比较直观地表示数据结构之间的关系，便于理解和实现

例题：一棵含有101个结点的二叉树中有36个叶子结点，度为 2 的结点个数是 \_和度为１的结点个数是 \_？（用方程求解101 - 1 = 36\*0 + y\*1 + x\*2）

·散列查找法（Hash Search）的思想，它通过对元素的关键字值进行某种运算，直接求出元素的地址，即使用关键字到地址的直接转换方法，而不需要反复比较。散列函数和散列地址：在记录的存储位置p和其关键字 key 之间建立一个确定的对应关系 H，使p=H(key)，称这个对应关系H为散列函数，p为散列地址。散列表：一个有限连续的地址空间，用以存储按散列函数计算得到相应散列地址的数据记录。通常散列表的存储空间是一个一维数组，散列地址是数组的下标。

·DFS回溯：每当一棵树或一个图向前移动并且现有路径上没有节点时，树就会沿着它前进的相同路径向后移动，以找到要遍历的新节点。这个过程不断迭代，直到所有未访问的节点都被访问过。

·图的表示：图可以用不同的数据结构来表示，包括邻接矩阵、邻接表、关联矩阵等。

图的遍历：图的遍历是指从图中的某个顶点出发，访问图中所有顶点且不重复的过程。常见的图遍历算法包括深度优先搜索（DFS：DFS算法的时间复杂度取决于图的顶点数和边数。在最坏情况下，每个节点和边都会被访问一次，因此时间复杂度为O(V + E)，其中V是顶点数，E是边数，空间复杂度为O(V)。eg骑士周游）和广度优先搜索（BFS会先探索当前深度图中的所有顶点，然后再移动到下一个深度级别的顶点。它从指定的顶点开始，访问其所有邻居，然后再移动到下一级邻居。BFS通常用于图中的路径查找、拓扑排序、二叉树的层序遍历、连通分量和最短路径问题的算法中。BFS算法的时间复杂度取决于图的顶点数和边数。在最坏情况下，每个节点和边都会被访问一次，因此时间复杂度为O(V + E)。）。

最短路径：最短路径算法用于找出两个顶点之间的最短路径，例如 Dijkstra 算法和 Floyd-Warshall 算法。

最小生成树：最小生成树算法用于在一个连通加权图中找出一个权值最小的生成树，常见的算法包括 Prim 算法和 Kruskal 算法。

拓扑排序：拓扑排序算法用于对有向无环图进行排序，使得所有的顶点按照一定的顺序排列，并且保证图中的边的方向符合顺序关系。拓扑排序在任务调度、依赖关系分析等领域有重要的应用。

图的连通性：图的连通性算法用于判断图中的顶点是否连通，以及找出图中的连通分量。

·更多图算法：

·最短路径算法：Dijkstra算法：用于找到两个顶点之间的最短路径；Bellman-Ford算法：用于处理带有负权边的图的最短路径问题；Floyd-Warshall算法：用于找到图中所有顶点之间的最短路径。

·最小生成树算法：Prim算法：用于找到连接所有顶点的最小生成树；Kruskal算法 / 并查集：用于找到连接所有顶点的最小生成树，适用于边集合已经给定的情况。

·拓扑排序算法：DFS：用于对有向无环图（DAG）进行拓扑排序；Karn算法 / BFS ：用于对有向无环图进行拓扑排序。

·强连通分量算法：Kosaraju算法 / 2 DFS：用于找到有向图中的所有强连通分量；Tarjan算法：用于找到有向图中的所有强连通分量。

Dijkstra算法与BFS（广度优先搜索）相似性：都是用于图的遍历，它们都从一个起始顶点开始，逐步扩展到邻居顶点，并以某种方式记录已经访问过的顶点。

不同：BFS是一种无权图的最短路径算法，它以层次遍历的方式遍历图，并找到从起始顶点到所有其他顶点的最短路径；Dijkstra算法（时间复杂度为O(V^2)，可以优化到O((V+E)logV)）是一种有权图的最短路径算法，它通过贪心策略逐步确定从起始顶点到所有其他顶点的最短路径。BFS使用队列来保存待访问的顶点，并按照顺序进行遍历。它不考虑权重，只关注路径的长度；Dijkstra算法使用优先队列（通常是最小堆）来保存待访问的顶点，并按照顶点到起始顶点的距离进行排序。它根据路径长度来决定下一个要访问的顶点，从而保证每次都是选择最短路径的顶点进行访问。

【BFS适用于无权图的最短路径问题，而Dijkstra算法适用于有权图的最短路径问题。】