**数据结构**

数据结构的核心三要素包括：

1. **逻辑结构：** 逻辑结构是指数据对象中数据元素之间的关系。常见的逻辑结构包括线性结构（如数组、链表）、树形结构（如二叉树、多叉树）、图形结构等。逻辑结构决定了数据元素之间的组织方式和相互关系。
2. **存储结构：** 存储结构是指数据结构在计算机内存中的表示方式。常见的存储结构包括顺序存储结构和链式存储结构。顺序存储结构利用连续的存储单元存储数据元素，适用于线性结构；链式存储结构利用指针将数据元素分散存储在内存中，适用于树形结构和图形结构。
3. **运算及算法：** 运算及算法是指在数据结构上定义的一组基本操作，以及在这些操作上实现的算法。常见的数据结构操作包括查找、插入、删除等。算法则是对这些操作的具体实现方式，通过选择合适的算法可以提高数据结构的效率。

以二叉搜索树为例解释这三个要素：

1. **逻辑结构：** 二叉搜索树是一种树形结构，其中每个节点最多有两个子节点，且左子节点的值小于等于父节点的值，右子节点的值大于等于父节点的值。这种有序的组织结构使得二叉搜索树具有快速的查找、插入和删除操作。
2. **存储结构：** 二叉搜索树通常使用链式存储结构。每个节点包含存储数据的域以及指向左子节点和右子节点的指针。这种存储方式使得二叉搜索树的节点可以动态分配内存，灵活性较高。
3. **运算及算法：** 二叉搜索树上常见的操作包括查找、插入和删除。其中，查找操作通过比较节点的值和目标值，根据二叉搜索树的有序性进行逐层搜索，以快速找到目标节点。插入操作将新节点按照有序性插入到合适的位置，保持二叉搜索树的有序性。删除操作则需要考虑节点的不同情况，包括删除叶子节点、删除只有一个子节点的节点以及删除有两个子节点的节点，并进行相应的调整以保持二叉搜索树的有序性。通过合适的算法实现这些操作，可以提高二叉搜索树的效率和性能。

**排序算法**

冒泡、选择、插入、希尔、快速、归并、堆

时间复杂度：

平均：

O（n^2）：冒泡、选择、插入、希尔

O（nlog(n)）: 快速、归并、堆

最坏：

O（n^2）：冒泡、选择、插入、希尔、快速

O（nlog(n)）: 归并、堆

空间复杂度：

O（1）：冒泡、选择、插入、希尔、堆

O（logn）:快速

O（n）:归并

能够保持稳定的：冒泡、插入、归并

**哈夫曼树**

**Q:** 用 Huffman 算法构造一个最优二叉编码树，待编码的字符权值分别为{3，4，5，6，8，9，11，12}，请问该最优二叉编码树的带权外部路径长度为（ B ）。（补充说明：树的带权外部路径长度定义为树中所有叶子结点的带权路径长度之和；其中，结点的带权路径长度定义为该结点到树根之间的路径长度与该结点权值的乘积） A：58 B：169 C：72 D：18

解释：为了构造哈夫曼树，我们遵循一个重复的选择过程，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。我们可以按照以下步骤操作：

1. 将给定的权值排序：{3, 4, 5, 6, 8, 9, 11, 12}。
2. 选择两个最小的权值：3 和 4，将它们组合成一个新的权值为 7 的节点。

现在权值变为：{5, 6, 7, 8, 9, 11, 12}。

1. 再次选择两个最小的权值：5 和 6，将它们组合成一个新的权值为 11 的节点。

现在权值变为：{7, 8, 9, 11, 11, 12}。

1. 选择两个最小的权值：7 和 8，将它们组合成一个新的权值为 15 的节点。

现在权值变为：{9, 11, 11, 12, 15}。

1. 选择两个最小的权值：9 和 11，将它们合并成一个新的权值为 20 的节点。

现在权值变为：{11, 12, 15, 20}。

1. 选择两个最小的权值：11 和 12，合并成一个新的权值为 23 的节点。

现在权值变为：{15, 20, 23}。

1. 选择两个最小的权值：15 和 20，合并成一个新的权值为 35 的节点。

现在权值变为：{23, 35}。

1. 最后，合并这两个节点得到根节点，权值为 23 + 35 = 58。

现在我们可以计算哈夫曼树的带权外部路径长度（WPL）。

**(58)**

**/ \**

**(23) (35)**

**/ \ / \**

**(11)(12) (20) (15)**

**/ \ / \**

**(9)(11) (7)(8)**

**/ \ / \**

**(5)(6)(3) (4)**

现在让我们计算每个叶子节点的带权路径长度：

* 权值 3 的节点路径长度为 4，WPL部分为 3 \* 4 = 12。
* 权值 4 的节点路径长度为 4，WPL部分为 4 \* 4 = 16。
* 权值 5 的节点路径长度为 4，WPL部分为 5 \* 4 = 20。
* 权值 6 的节点路径长度为 4，WPL部分为 6 \* 4 = 24。
* 权值 9 的节点路径长度为 3，WPL部分为 9 \* 3 = 27。
* 权值 8 的节点路径长度为 3，WPL部分为 8 \* 3 = 24。
* 权值 11 的节点路径长度为 2，WPL部分为 11 \* 2 = 22。
* 权值 12 的节点路径长度为 2，WPL部分为 12 \* 2 = 24。

将所有部分的 WPL 相加，我们得到整棵哈夫曼树的 WPL：

WPL = 12 + 16 + 20 + 24 + 27 + 24 + 22 + 24 = 169

**堆**

首先实现二叉堆的构造方法。既然用一个列表就可以表示整个二叉堆，那么构造方法要做的就是初始化这个列表与属性 currentSize，用于记录堆的当前大小。代码清单 6-17 给出了构造方法的 Python 代码。列表 heapList 的第一个元素是 0，它的唯一用途是为了使后续的方法可以使用整数除法。

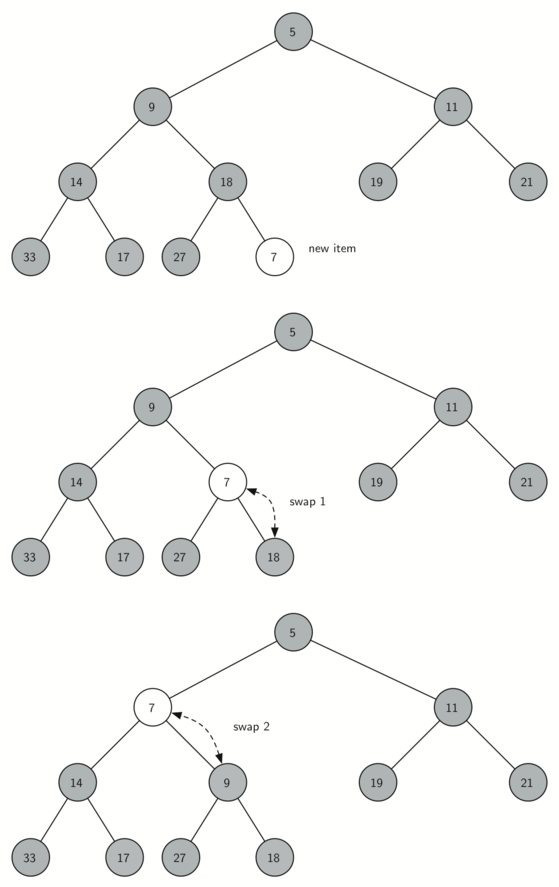
class BinHeap:

def \_\_init\_\_(self):

self.heapList = [0]

self.currentSize = 0

接下来实现insert方法。将元素加入列表的最简单、最高效的方法就是将元素追加到列表的末尾。追加操作的优点在于，它能保证完全树的性质，但缺点是很可能会破坏堆的结构性质。不过可以写一个方法，通过比较新元素与其父元素来重新获得堆的结构性质。如果新元素小于其父元素，就将二者交换。图3展示了将新元素放到正确位置上所需的一系列交换操作。



注意，将元素往上移时，其实是在新元素及其父元素之间重建堆的结构性质。此外，也保留了兄弟元素之间的堆性质。当然，如果新元素很小，需要继续往上一层交换。代码清单6-18给出了percUp方法的代码，该方法将元素一直沿着树向上移动，直到重获堆的结构性质。此时，heapList中的元素0正好能发挥重要作用。我们使用整数除法计算任意节点的父节点。就当前节点而言，父节点的下标就是当前节点的下标除以2。

代码清单6-18 percUp方法

def percUp(self,i):

while i // 2 > 0:

if self.heapList[i] < self.heapList[i // 2]:

tmp = self.heapList[i // 2]

self.heapList[i // 2] = self.heapList[i]

self.heapList[i] = tmp

i = i // 2

现在准备好编写insert方法了。代码清单6-19给出了该方法的Python代码。其实，insert方法的大部分工作是由percUp方法完成的。当元素被追加到树中之后，percUp方法将其移到正确的位置。

代码清单6-19 向二叉堆中新加元素

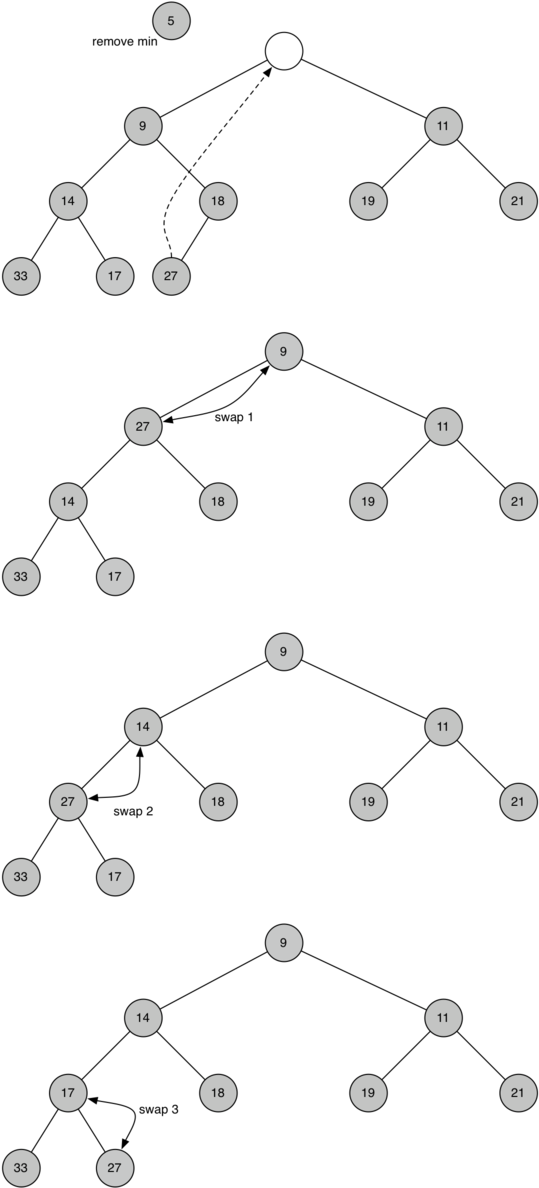
def insert(self,k):

self.heapList.append(k)

self.currentSize = self.currentSize + 1

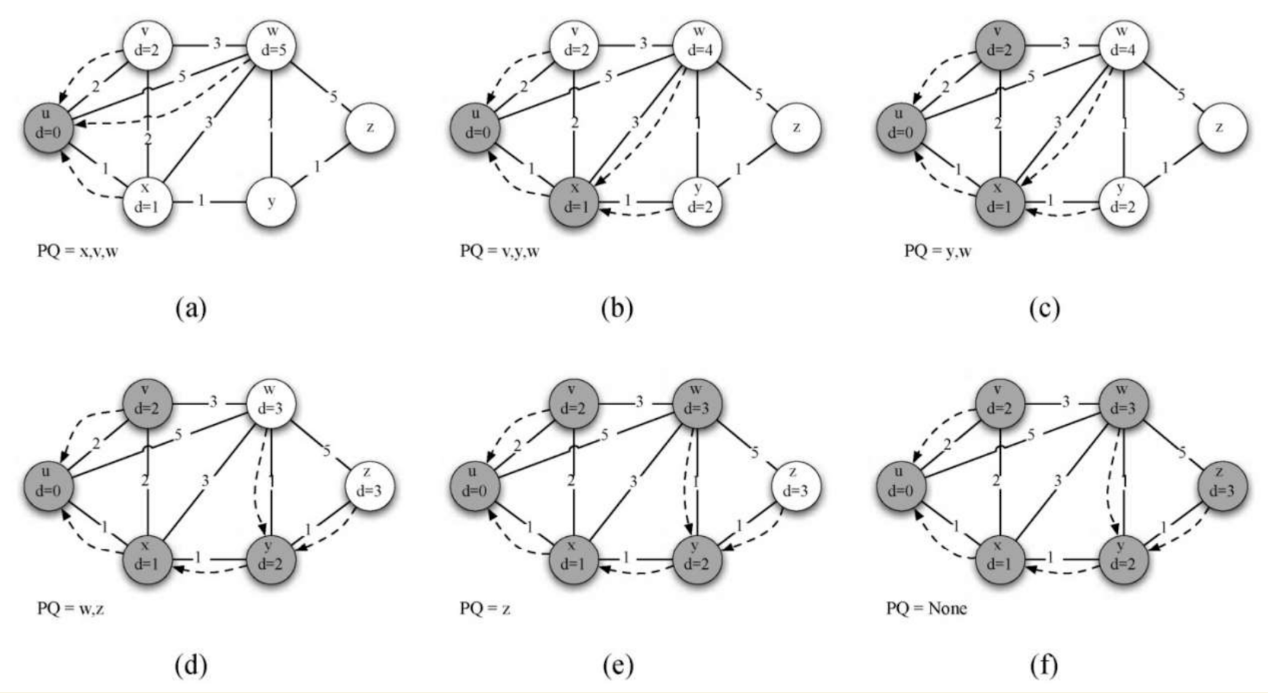
self.percUp(self.currentSize)

正确定义insert方法后，就可以编写delMin方法。既然堆的有序性质要求根节点是树的最小元素，那么查找最小值就很简单。delMin方法的难点在于，如何在移除根节点之后重获堆的结构性质和有序性。可以分两步重建堆。第一步，取出列表中的最后一个元素，将其移到根节点的位置。移动最后一个元素保证了堆的结构性质，但可能会破坏二叉堆的有序性。第二步，将新的根节点沿着树推到正确的位置，以重获堆的有序性。图4展示了将新的根节点移动到正确位置所需的一系列交换操作。



**Dijkstra 算法**

让我们对照图3 来理解如何针对每一个顶点应用Dijkstra算法。从顶点u开始，与u相邻的3个顶点分别是v、w和x。由于到v、w和x的初始距离都是sys.maxint，因此从起点到它们的新开销就是直接开销。更新这3个顶点的开销，同时将它们的前驱顶点设置成 u，并将它们添加到优先级队列中。我们使用距离作为优先级队列的键。此时，算法运行的状态如图3a所示。



下一次while循环检查与x相邻的顶点。之所以x是第2个被访问的顶点，是因为它到起点的开销最小，因此排在了优先级队列的头部。与x相邻的有u、v、w和y。对于每一个相邻顶点，检查经由x到它的距离是否比已知的距离更短。显然，对于y来说确实如此，因为它的初始距离是sys.maxsize；对于u和v来说则不然，因为它们的距离分别为0和2。但是，我们发现经过x到w的距离比直接从u到w的距离要短。因此，将到达w的距离更新为更短的值，并且将w的前驱顶点从u改为x。图3b展示了此时的状态。

下一步检查与v相邻的顶点。这一步没有对图做任何改动，因此我们继续检查顶点y。此时，我们发现经由y到达w和z的距离都更短，因此相应地调整它们的距离及前驱顶点。最后检查w和z，发现不需要做任何改动。由于优先级队列为空，因此退出。

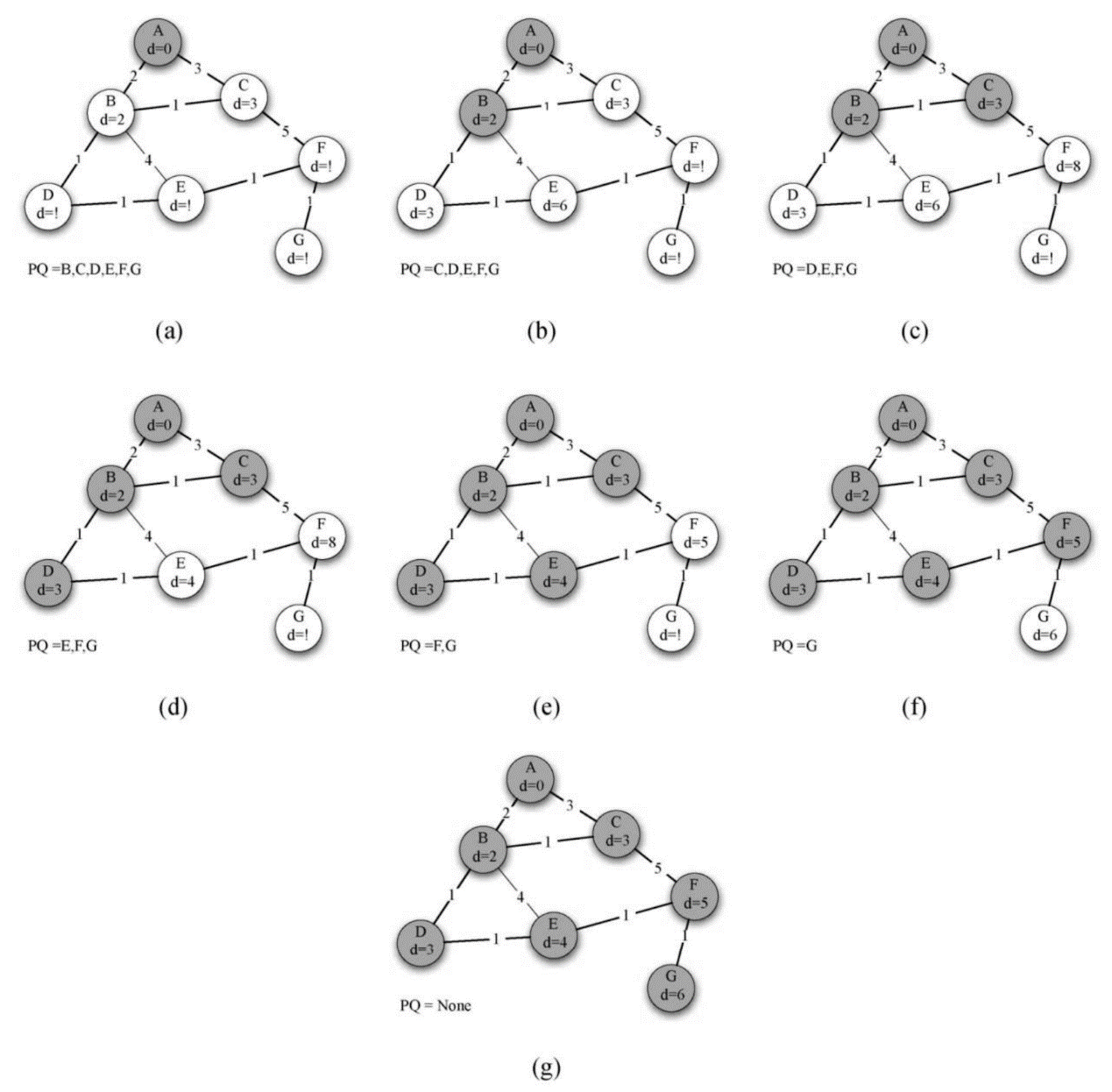
非常重要的一点是，Dijkstra算法只适用于边的权重均为正的情况。如果图2中有一条边的权重为负，那么Dijkstra算法永远不会退出。

**Prime算法**

图3展示了将Prim算法应用于示例生成树的过程。以顶点A作为起点，将A到其他所有顶点的距离都初始化为无穷大。检查A的相邻顶点后，可以更新从A到B和C的距离，因为实际的距离小于无穷大。更新距离之后，B和C被移到优先级队列的头部。并且，它们的前驱顶点被设置为A。注意，我们还没有把B和C添加到生成树中。只有在从优先级队列中移除时，顶点才会被添加到生成树中。

由于到B的距离最短，因此接下来检查B的相邻顶点。检查后发现，可以更新D和E。接下来处理优先级队列中的下一个顶点C。与C相邻的唯一一个还在优先级队列中的顶点是F，因此更新到F的距离，并且调整F在优先级队列中的位置。

现在检查与D相邻的顶点，发现可以将到E的距离从6减少为4。修改距离的同时，把E的前驱顶点改为D，以此准备将E添加到生成树中的另一个位置。Prim算法正是通过这样的方式将每一个顶点都添加到生成树中。



**二叉搜索树**

快速排序是一种基于分治法的排序算法，它通过选择一个元素作为基准（pivot），将数组分割为两个子数组，其中一个子数组的元素都小于基准，另一个子数组的元素都大于基准。然后，对两个子数组递归地应用相同的排序过程，直到排序完成。

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。

可以使用二叉搜索树来实现快速排序的过程。具体步骤如下：

1. 选择数组中的一个元素作为基准。
2. 创建一个空的二叉搜索树。
3. 将数组中的其他元素逐个插入二叉搜索树中。
4. 按照二叉搜索树的中序遍历（左子树、根节点、右子树）得到排序后的结果。

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。

需要注意的是，二叉搜索树的性能取决于树的平衡性。如果二叉搜索树变得不平衡，性能可能会下降到 O(n^2) 的时间复杂度。因此，在实际应用中，为了确保性能，通常会使用平衡二叉搜索树（如红黑树、AVL树）来实现快速排序。

**树转换为二叉树**

森林转化为二叉树的方法通常包括以下几个步骤：

1.将每棵树转换为二叉树。这一步涉及在所有兄弟结点之间加一条连线，然后对树中的每个结点，只保留它与第一个孩子结点之间的连线，删除它与其他孩子结点之间的连线。

2.连接二叉树。从第二棵二叉树开始，依次把后一棵二叉树的根结点作为前一棵二叉树根结点的右孩子结点，用线连接起来。

**拓扑排序**

**4.1.3 Kahn算法 / BFS**

Kahn算法是基于广度优先搜索（BFS）的一种拓扑排序算法。

Kahn算法的基本思想是通过不断地移除图中的入度为0的顶点，并将其添加到拓扑排序的结果中，直到图中所有的顶点都被移除。具体步骤如下：

1. 初始化一个队列，用于存储当前入度为0的顶点。
2. 遍历图中的所有顶点，计算每个顶点的入度，并将入度为0的顶点加入到队列中。
3. 不断地从队列中弹出顶点，并将其加入到拓扑排序的结果中。同时，遍历该顶点的邻居，并将其入度减1。如果某个邻居的入度减为0，则将其加入到队列中。
4. 重复步骤3，直到队列为空。

Kahn算法的时间复杂度为O(V + E)，其中V是顶点数，E是边数。它是一种简单而高效的拓扑排序算法，在有向无环图（DAG）中广泛应用。

如果 result 列表的长度等于图中顶点的数量，则拓扑排序成功，返回结果列表 result；否则，图中存在环，无法进行拓扑排序。

#### Kosaraju

Kosaraju算法是一种用于在有向图中寻找强连通分量（Strongly Connected Components，SCC）的算法。它基于深度优先搜索（DFS）和图的转置操作。

Kosaraju算法的核心思想就是两次深度优先搜索（DFS）。

1. **第一次DFS**：在第一次DFS中，我们对图进行标准的深度优先搜索，但是在此过程中，我们记录下顶点完成搜索的顺序。这一步的目的是为了找出每个顶点的完成时间（即结束时间）。
2. **反向图**：接下来，我们对原图取反，即将所有的边方向反转，得到反向图。
3. **第二次DFS**：在第二次DFS中，我们按照第一步中记录的顶点完成时间的逆序，对反向图进行DFS。这样，我们将找出反向图中的强连通分量。

Kosaraju算法的关键在于第二次DFS的顺序，它保证了在DFS的过程中，我们能够优先访问到整个图中的强连通分量。因此，Kosaraju算法的时间复杂度为O(V + E)，其中V是顶点数，E是边数。