1. Sorting

1.1 O(n^2) Sorting

Bubble Sort

```
def BubbleSort(arr):
        n = len(arr)
    # 遍历所有数组元素
    for i in range(n): # O(n)
        # 标记此轮遍历是否进行了交换
        swapped = False
        for j in range(0, n - i - 1): # O(n)
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True
        if not swapped:
            break
```

- 迭代总数:$N \times (N-1)/2$
- 改进思路：如果我们在第二个循环中完全没有交换，则意味着数组已经排序完毕，此时可以停止Bubble Sort

Selection Sort

```
def selection_sort(arr):
    # 遍历所有数组元素
    for i in range(len(arr)):
        # 找到剩余未排序元素中的最小值
        min_idx = i
        for j in range(i+1, len(arr)):
            if arr[min_idx] > arr[j]:
                min_idx = j

        # 将找到的最小元素交换到当前遍历的起始位置
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
```

Insertion Sort

```
def insertionSort(array A, integer N):
        for each element i in [1,...,N-1]:
                x = A[i]
                for j from 0 to i-1:
                        if A[j] > x:
                                A[j+1] = A[j]
                        else: break
                A[j+1] = x
```

1.2 O(NlogN) Sorting

Merge Sort

```
def merge_sort(arr):
    if len(arr) > 1:
        # 寻找中点，进行分解
        mid = len(arr) // 2
        L = arr[:mid]  # 获取左半部分
        R = arr[mid:]  # 获取右半部分

        # 递归地对半分进行分解
        merge_sort(L)
        merge_sort(R)
        i = j = k = 0
```

```
        # 合并过程
        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1
        # 检查是否有任何元素遗留在 L[] 和 R[] 中
        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1
        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1
```

Quick Sort

- 选择支点P
- 将数组A[i,…,j]分为三部分,A[i,…,m-1],A[m],A[m+1,…,j]
  - A[i,…,m-1]包含小于或等于p的项目
  - A[m]=p,即索引m是p在数组A排序中的正确位置
  - A[m+1,…,j]包含大于或等于p的项目
- 对这两部分进行递归排序

```
import random

def partition(A, i, j):
    p = A[i]  # p is the pivot
    m = i  # S1 and S2 are initially empty
    for k in range(i+1, j+1):  # explore the unknown region
        if (A[k] < p) or (A[k] == p and random.randint(0, 1) == 0):  # case 2+3
            m += 1
            A[k], A[m] = A[m], A[k]  # exchange these two indices
        # notice that we do nothing in case 1: A[k] > p
    A[i], A[m] = A[m], A[i]  # final step, swap pivot with A[m]
    return m  # return the index of pivot

def quickSort(A, low, high):
    if low < high:
        m = partition(A, low, high)  # O(N)
        # A[low..high] ~> A[low..m-1], pivot, A[m+1..high]
        quickSort(A, low, m-1)  # recursively sort left subarray
        # A[m] = pivot is already sorted after partition
        quickSort(A, m+1, high)  # then sort right subarray
```

O(N) Sorting

Counting Sort

- 假设：如果要排序的项目是范围较小的整数，我们可以计算每个整数（在该小范围内）出现的频率，然后在该小范围内循环，按排序顺序输出项目。
- 计算频率的时间复杂度为 O(N)，按排序顺序打印输出的时间复杂度为 O(N+k)，其中 k 是输入整数的范围。因此，计数排序的时间复杂度为 O(N+k)，如果 k 较小，则为 O(N)

Radix Sort

- 假设如果要排序的项目是范围大但位数少的整数，我们可以将计数排序思想与阶乘排序结合起来，以实现线性时间复杂度。
- 在 Radix 排序中，我们将每个要排序的项目视为 w 位数的字符串（必要时，我们会将小于 w 位数的整数用前导零填充）。

- 基数排序不是通过比较数值来排序，而是通过按数字级别分配桶来实现排序，每个级别的排序可以采用不同的排序算法。一般情况下，对于十进制数，就会基于数位来进行排序，先从最低有效数字（比如个位）开始，一直到最高有效数字。

2. Linked List

2.1 Array

- 紧凑数组compact array是实现List的理想选择
  - 紧凑指的是没有空隙
- `get(i)`: $O(1)$, 返回索引为i的元素
- `search(v)`: $O(N)$ (最好情况为$O(1)$)，搜索某个值v是否在数组中
- `insert(i,v)`: $O(N)$ (当在数组末尾插入时为$O(1)$)，插入值v到索引i
  - 当我们插入到除了末尾之外的位置时，需要将后续的元素后移
- `remove(i)`: $O(N)$ (当在数组末尾删除时为$O(1)$)，在索引i处删除元素
  - 当我们删除末尾之外的位置时，需要将后续元素前移

2.2 Linked List

```
struct Vertex { // we can use either C struct or C++/Python/Java class
  int item; // the data is stored here, an integer in this example
  Vertex* next; // this pointer tells us where is the next vertex
};
```

- `get(i)`: $O(N)$，因为链表只保留了头部和尾部指针，访问中间的元素需要从头遍历
- `search(v)`: $O(N)$ (最好情况为$O(1)$)，搜索某个值v是否在数组中
- `insert(i,v)`: 插入值v到索引i
  - 在头部插入(i=0): $O(1)$
  - 插入一个空列表(i=0): $O(1)$
  - 在中间插入($i \in [1, \ldots, N-1]$): $O(N)$
  - 在尾部插入(i=N): $O(1)$
- `remove(i)`: $O(N)$ (当在数组头部删除时为$O(1)$)，在索引i处删除元素

Doubly Linked List

- 现在可以在$O(1)$中删除末尾元素

2.3 Stack

- 后进先出
- `pop()`: 弹出最上层的元素,$O(1)$
- `push()`: 压入元素,$O(1)$

2.4 Queue/Dequeue

- 先进先出
- 对于队列，可以查看头部元素，在末尾插入元素，在头部移除元素
- 对于双端队列(deque)，可以：
  - `peek(head/tail)`：查看头部或末尾的元素
  - `insert(head/tail)`：在头部或末尾插入元素
  - `remove(head/tail)`: 在头部或末尾移除元素
  - 所有操作均为$O(1)$

Queue/Deque in Python

- `from collection import deque`: 在Python中只集成了deque双端队列。

- **append(x)**：在右侧添加一个元素。$O(1)$
- **appendleft(x)**：在左侧添加一个元素。$O(1)$
- **pop()**：移除并返回右侧的一个元素。$O(1)$
- **popleft()**：移除并返回左侧的一个元素。$O(1)$
- **extend(iterable)**：在右侧添加多个元素。$O(k)$,k是迭代器中元素的数量
- **extendleft(iterable)**：在左侧添加多个元素（注意添加的顺序是反转的）。$O(k)$
- **rotate(n)**：向右旋转队列 `n` 步。如果 `n` 是负数，则向左旋转。$O(k)$
  - 当 `n > 0` 时，`deque` 的最右侧的 `n` 个元素会被移动到队列的左侧。
  - 当 `n < 0` 时，`deque` 的最左侧的 `-n` 个元素会被移动到队列的右侧。
  - 当 `n = 0` 时，不会发生任何变化。

```
from collections import deque

dq = deque([1, 2, 3, 4, 5])
dq.rotate(1)
print(dq)  # 输出: deque([5, 1, 2, 3, 4])
dq.rotate(-1)
print(dq)  # 输出: deque([1, 2, 3, 4, 5])
dq.rotate(2)
print(dq)  # 输出: deque([3, 4, 5, 1, 2])
```

- **clear()**：清空队列。$O(n)$
- **count(x)**：计算队列中元素 `x` 的个数。$O(n)$
- **index(x, [start, [stop]])**：返回元素 `x` 在队列中的索引。$O(n)$
- **insert(i, x)**：在位置 `i` 插入元素 `x`。$O(n)$
- **remove(x)**：移除队列中第一个匹配的元素 `x`。$O(n)$

3. Binary (Max) Heap

- 最大二叉堆是一种特殊的**完全二叉树**，用于模拟**优先队列**。
  - **完全二叉树Complete Binary Tree**：完整二叉树二叉树中的每一级（可能是最后一级/最低一级除外）都被完全填满，最后一级的所有顶点都**尽可能靠左**
  - **优先队列Priority Queue**: 优先队列是一种特殊的队列，其中每个元素都有优先级。元素的添加是无序的，但是元素的删除是根据优先级进行的，优先级最高的元素首先被移除。
    - `Enqueue`: 向队列添加一个元素。插入操作不考虑优先级，只是简单地添加元素到队列中
    - `Dequeue`: 移除优先级最高的元素。如果有多个元素具有相同的最高优先级，则根据队列的具体实现，可以选择任何一个
    - `Peek`: 查看优先级最高的元素，但不从队列中移除它
- 在最大二叉堆中，每个节点的值都**大于或等于**其子节点(child node)的值。这意味着堆的根节点(root node)总是最大的元素
- 对于包含N个元素的二叉堆，他的高度不会超过log_2N，因为这是一个完全二叉树

3.1 (最大)二叉堆操作

- `Create(A)`: $O(NlogN)$, 调用多次`Insert(v)`, 也有$O(N)$版本
- `Insert(v)`: $O(logN)$
  - 在二叉最大堆中插入新项目 v 时，只能在最后一个索引 N 加 1 处进行，以保持紧凑数组 = 完整二叉树属性。然而，最大堆属性仍可能被违反。该操作将从插入点向上修复最大堆属性（如有必要），并在不再违反最大堆属性时停止。
- `ExtractMax()`: $O(logN)$
  - 此操作提出并删除二叉堆中最大的元素(根节点)，并使用现有元素替代(为最后一个索引N)，然而此操作从根节点向下修复最大堆属性
- `UpdateKey(i, new_v)`: $O(logN)$ 如果i已知
  - 需要向上和向下修复最大堆属性
- `Delete(i)`: $O(logN)$ 如果i已知

3.2 Python中的二叉堆

- `import heapq`：`heapq`提供了最小堆的实现，如果需要实现最大堆，则可以将元素取反来实现。
- `heapq.heappush(heap, item)`：将元素 `item` 添加到堆 `heap` 中。这会保持堆的不变性，即堆的第一个元素始终是最小的。$O(logn)$
- `heapq.heappop(heap)`：弹出并返回 `heap` 中的最小元素，同时保持剩余元素的堆不变性。$O(logn)$
- `heapq.heappushpop(heap, item)`：将 `item` 放入堆中，然后弹出并返回堆中的最小元素。这个操作比单独调用 `heappush` 和 `heappop` 更有效率。$O(logn)$
- `heapq.heapify(x)`：将列表 `x` 转换成堆，即重新排列列表 `x` 的元素，使其符合堆的性质。这是以线性时间运行的，非常高效$O(n)$

3.3 Heap Sort

堆排序（Heap Sort）是一种基于比较的排序算法，它利用堆这种数据结构来实现。堆是一种特殊的完全二叉树，其中每个节点的值都大于或等于（最大堆）或小于或等于（最小堆）其子节点的值。堆排序算法可以分为两个主要步骤：

1. 建立堆：将待排序的数组构建成一个最大堆或最小堆。构建堆的过程是从最后一个非叶子节点开始，逐步向上调整，确保每个子树都满足堆的性质。
2. 排序过程：
   - 在最大堆中，堆的根节点是最大的元素。将其与堆的最后一个元素交换，然后减小堆的大小，并对新的根节点进行调整，使其满足堆的性质。这个过程会将当前最大的元素放置在数组的正确位置。
   - 重复上述步骤，直到堆的大小变为1，此时整个数组变为有序。

堆排序的特点

- 时间复杂度：堆排序的时间复杂度为 $O(nlogn)$，其中 n 是数组的大小。这是因为创建堆的时间复杂度为 $O(n)$，而每次调整堆的时间复杂度为 $O(logn)$，总共需要进行 n-1 次调整。
- 空间复杂度：堆排序是原地排序算法，空间复杂度为 $O(1)$。
- 不稳定排序：堆排序是不稳定的排序算法，即相同的元素可能在排序后改变其相对位置。
- 适用场景：堆排序适用于大数据量的排序，特别是在需要找出最大或最小元素时效率较高。

4. Binary Search Tree

4.1 BST & AVL Tree

二叉搜索树（BST）

二叉搜索树是一种有以下性质的二叉树：

1. 每个节点的值都大于其左子树上任意节点的值。
2. 每个节点的值都小于其右子树上任意节点的值。
3. 左右子树也分别是二叉搜索树。

BST的优点是实现简单，提供了有效的查找、插入和删除操作。然而，它的主要缺点是不保证树的平衡。在最坏的情况下（例如，连续插入有序的数据），BST可以退化成一个链表，导致查找、插入和删除操作的时间复杂度变为O(n)。

AVL树

AVL树是BST的一种改进，得名于其发明者Adelson-Velsky和Landis。AVL树在BST的基础上增加了额外的平衡条件：

- AVL树是一种高度平衡的二叉搜索树。
- 每个节点的左右子树的高度差（平衡因子）最多为1。

为了维护这种高度平衡，AVL树在插入和删除节点时可能需要通过旋转来重新平衡。这些旋转操作有助于确保树的高度大致保持在log(n)，从而使得查找、插入和删除操作的时间复杂度都维持在O(log n)。

主要区别

1. 平衡性：AVL树保证了树的平衡，而普通的BST不保证。

2. 性能：在AVL树中，查找、插入和删除操作的时间复杂度稳定为O(log n)，而在BST中这些操作的时间复杂度在最坏情况下可能达到O(n)。
3. 实现复杂度：由于维护平衡的需要，AVL树的实现比普通BST更复杂。
4. 内存占用：AVL树由于存储平衡因子或高度信息，通常会比普通BST占用更多的内存。
5. 旋转操作：AVL树在插入和删除时可能需要进行旋转操作以维护树的平衡，这是普通BST所不需要的。

4.2 BST

- BST的每个节点都有：
  - 指向左侧和右侧子节点的指针
  - 指向父节点的指针
  - 键值数据
  - 每个键的频率

BST Query Operation (remain BST Structure)

- `search(v)`: 我们设置当前节点为根，然后检查当前顶点小于/等于/大于我们需要搜索的整数V，然后分别转到右侧/停止/左侧子树，直到我们找到节点
- `searchMin(), searchMax()`: 从根开始向左/右侧子树查找最小/最大元素
- `successor(v)`：寻找整数v的后继数。如果v没有右子树，则找到第一个大于顶点v的父顶点w。如果有右子树，则找到右子树中最小的节点。
- `predecessor(v)`: 寻找整数v的前任数。如果v没有左子树，则找到第一个小于顶点v的父顶点w。如果有左子树，则找到左子树中最大的节点
- 这些操作的时间复杂度为$O(h)$,h为BST的高度
  - BST的最大高度可能为N

BST Update Operation (change BST structure)

- `insert(v)`: 从根节点出发，如果比该节点大，则转到右子树，如果比该节点小则左子树。以此类推，直到左或右子树没有节点，此时插入在此位置
- `remove(v)`: 三种情况
  - 第一种情况：顶点 v 目前是 BST 的叶顶点(leaf vertex, 即没有子节点的节点)之一。我们只需删除该叶顶点
  - 第二种情况：顶点 v 是 BST 的（内部/根）顶点，并且它正好有一个子顶点。删除 v 而不做任何其他操作将断开 BST 的连接。我们可以将该顶点的唯一子顶点与该顶点的父顶点连接起来.
  - 第三种情况：顶点 v 是 BST 的一个（内部/根）顶点，并且它正好有两个子顶点。删除 v 而不做任何其他操作将断开 BST 的连接。我们用它的后继顶点替换该顶点。由于需要找到后继顶点，这部分需要 O(h) 的时间
- 这些操作的时间复杂度为$O(h)$,h为BST的高度
  - BST的最大高度可能为N

Bound of BST Height

- BST的高度下限为log_2N, 上限为N

5. HashTable

5.1 Table ADT

- 表结构通常至少有三种操作：
  - `search(v)`：确定v是否在表中
  - `insert(v)`：将v插入表中
  - `remove(v)`：将v从表中移除

5.2 Direct Addressing Table

- 直接寻址表适用于整数范围小的情况。对于一个需要存储M个值的情况，我们使用大小为M的空数组A。

- 我们使用要存储的键值本身来确定其在数组A中的地址。因此被称为直接寻址。
- 在直接寻址中，三个表操作(search, insert, remove)均为$O(1)$

5.3 HashTable

- 使用一个哈希函数`h()`来将一串数字映射为一个较小范围的数字
- 通常一个哈希函数为`h(v) = v % M`，因此我们需要准备一个大小为M的空数组
- 插入时，我们插入元素的原始数值，如果需要插入卫星数据(satellite data)，可以使用set(v, satellite-data)
- 哈希函数实现的是多对一函数，即一个哈希值可以对应多个不同数值的元素

选择一个好的哈希函数值M

目标

- 尽量避免哈希碰撞
- 元素在每个哈希值之间分配均匀（不会出现某个哈希值slot有过多的元素）

选择方式

- 选择一个尽可能大的素数
- 不要靠近2的幂
- load factor < 0.5 (load factor = N/M)
  按照优先级排列，即优先满足第一个条件

5.4Collision Resolution

Closed Addressing

Separate Chaining

- 对于一个大小为M的哈希表，我们准备M份辅助数据结构（双向链表），如果两个键a和b有相同的哈希值，则这两个键都被追加到所在哈希值索引的双向链表中。
- 对于此技术，load factor可以大于1，load factor取决于M个双向链表的平均长度，这决定了搜索V的性能

Open Addressing (OA)

- 在此解决方法中，每个slot只能存储一个元素，我们使用以下符号表示一些重要概念
- M = HT.length = current hash table size(length)
- base = (key%M)
- step = the current probing step
- secondary = smaller_prime - key%smaller_prime
- 此方法中有三种技术：Linear Probing (LP), Quadratic Probing (QP), Double Hashing (DH)

Linear Probing

- 当出现碰撞（当前哈希值中存在元素），线性探测LP会往前继续搜索可用的插槽，当扫描到最后一个插槽时，会从头开始
- 当我们删除一个元素时，我们设定该元素所在的索引为DEL，此可以被未来的搜索函数绕过，但可以被insert函数覆盖，这被称为lazy deletion

聚类问题 Primary Clustering

- Cluster簇被定义为连续占用的插槽集合，覆盖当前键的基地址的簇被称为主簇primary cluster
- 线性探测可能会产生一个大型的主簇，这会导致搜索/插入/删除的时间远远大于$O(1)$

6. Graph

6.1 Read data from list (into dict)

```
language_index = {lang: idx for idx, lang in enumerate(languages, start=1)}
```

- 对于输入的index不是数字的情况下

6.2 Graph Data Structures

- 三种图数据结构适用于不同的图:
  - AM:
    - 稠密图, 即边的数量约等于节点数量的平方
    - 适合频繁查询两个节点间是否存在边的情况
    - 空间复杂度 $O(n^2)$
  - AL:
    - 适合稀松图，即边的数量远小于节点数量的平方
    - 适合频繁查询节点的相邻节点的情况
    - 空间复杂度 $\approx 2e$
    - For limited RAM
  - EL:
    - 适合较小数量的边
    - For limited RAM
    - 适合简单的图
    - 空间复杂度 e
    - 适合频繁检索（排序）所有的边的情况

Adjacency Matrix

- 邻接矩阵(AM)是一个正方形的矩阵，其中`AM[i][j]`表示从节点i到节点j的边的权重
  - 通常设置`AM[i][j]=0`来表示没有从i到j的边，然而如果图中包含值为0的加权边，则需要用其他符号(-1, None, NULL)表示无边
- 使用V*V的二维数组来实现此结构
- 空间复杂度为$O(V^2)$

```
def create_adjacency_matrix(n, edges):
    adj_matrix = [[float('inf') for _ in range(n)] for _ in range(n)]
    for i in range(n):
        adj_matrix[i][i] = 0
    for edge in edges:
        l1, l2, cost = edge
        adj_matrix[l1][l2] = cost
        adj_matrix[l2][l1] = cost
    return adj_matrix
```

Adjacency List

- 邻接表(AL)是一个由V个列表组成的数组，每个节点有一个列表。对于每个顶点i,`AL[i]`存储i的邻居，可以存储(neighbour number, weight)对
- 使用向量对向量实现这种数据结构`AL = [[] for _ in range(N)]`
- 空间复杂度为$O(V+E)$，比AM要高效的多

```
def create_adjacency_list(n, edges):
    adj_list = [[] for _ in range(n)]
    for edge in edges:
        l1, l2, cost = edge
        adj_list[l1].append((l2, cost))
        adj_list[l2].append((l1, cost))
    return adj_list
```

Edge List

- 边列表(EL)是边的集合，包括边所链接的节点和其权重。通常这些边是按权重递增排列的。
- 使用数组来实现`EL=[]`
- 空间复杂度为$O(E)$，比AL要高效的多

```
def create_edge_list(edges):
    edge_list = []
    for edge in edges:
        l1, l2, cost = edge
        edge_list.append((l1, l2, cost))
    return edge_list
```

7. Graph Traversal

7.1 DFS

- DFS takes one input param: the source vertex `s`
- Trying all options:
  1. If DFS is at a vertex `u` and it has `X` neighbours, it will pick the first neighbour V_1 (usually the vertex with the smallest vertex number)
  2. Recursively explore all reachable vertices from vertex V_1, and backtrack to vertex u
  3. Do the same process (step 1 and 2) for all neighbours, until it finishes exploring the last neighbour V_x and its reachable vertices
- Use an array `status[u]` to record the vertex u is visited or not
- Use another array `p[u]` of size V vertices to remember the parent/predecessor/previous of each vertex u along the DFS traversal path
  - For source vertex s, `p[s]` is set to `-1` since the source vertex has no predecessor
- Time complexity is $O(V+E)$
  - Each vertex is only visited once due to the fact that DFS will only recursively explore a vertex u if `status[u] = unvisited`, which takes $O(V)$ time
  - Every time a vertex is visited, all its k neighbours are explored and therefore after all vertices are visited, we have examined all E edges, which takes $O(E)$ time

1. 从一个选定的源节点开始，将其标记为“已访问”，并将其放入栈中。
2. 取栈顶元素为当前节点，探索当前节点的一个未访问的邻居节点。
3. 将新发现的节点标记为“已访问”并放入栈中。
4. 如果当前节点没有未访问的邻居节点，则将它从栈中弹出（回溯）。
   - 这意味着如果重复步骤，则在步骤2选定的节点为该节点的上一个节点
5. 重复步骤2到4，直到栈为空，或者找到目标节点，或者遍历完所有可达的节点。

DFS算法的特点是尽可能深地搜索树的分支，这个特性使得DFS在某些情况下比广度优先搜索（BFS）更加高效，比如在寻找解决方案的路径比较深的问题中。此外，DFS算法的空间复杂度通常较低，因为它不需要存储所有层的节点，只需要维护一个栈的空间即可。

```
def dfs(graph, start, visited=None):
        if visited is None:
                visited = set()
        visited.add(start)

        print(start, end=' ')
        for next in graph[start] - visited:
                dfs(graph, next, visited)
        return visited

# 示例图
graph = {
        'A': set(['B', 'C']),
```

```
        'B': set(['A', 'D', 'E']),
        'C': set(['A', 'F']),
        'D': set(['B']),
        'E': set(['B', 'F']),
        'F': set(['C', 'E'])
}
```

7.2 BFS

- BFS take one input param: the source vertex `s`
- BFS start from a source vertex `s` but is uses a queue to order the vistation sequence as *breadth* as possible before going deeper
- BFS also uses a Boolean array of size V vertices to distinguish between two states: visited and unvisited vertices
- Time complexity is $O(V+E)$
  - Each vertex is only visited once as it can only enter the queue once, which takes $O(V)$ time
  - Every time a vertex is dequeued from the queue, all its k neighbours are explored and therefore after all vertices are visited, we examined all E edges, which takes $O(E)$ time

1. 初始化：首先将根节点放入队列中。
2. 循环遍历：只要队列不为空，就重复以下步骤：
   - 从队列的前端取出一个节点。
   - 检查它是否为目标。如果找到目标，则搜索结束。
   - 如果它不是目标，则将该节点的所有未访问的邻接点加入队列，并标记这些邻接点为已访问。
3. 访问节点：对于队列中的每个节点，访问该节点，并检查它是否是目标节点。如果是，则结束搜索并返回结果。如果不是，则将其所有未被访问过的邻居节点加入队列。
4. 标记已访问：在加入队列的同时，应该将节点标记为已访问，以防止将节点重复加入队列。
5. 重复：重复步骤2，直到找到目标节点或队列为空，队列为空意味着整个图已经搜索完毕，没有找到目标。

BFS算法的特点在于它提供了最短路径的保证，即第一次找到目标节点时的路径是从根节点到目标节点的最短路径。这个特性使得BFS特别适合解决最短路径或最少步骤的问题。

```
from collections import deque

def bfs(graph, start):
        visited = set()
        queue = deque([start])

        while queue:
                vertex = queue.popleft()
                if vertex not in visited:
                        visited.add(vertex)
                        print(vertex, end=' ')
                        queue.extend(graph[vertex] - visited)

        return visited
```

7.3 Applications

Reachability test

- Call `DFS(s)` or `BFS(s)` to check if `status[t] = visited`

Print the Traversal Path

- For DFS, print each node when a node is marked as visited
  - i.e., print node once explore to it
- For BFS, print each node when dequeue a node
  - i.e., print node once it is get out form the queue to explore

Identifying a Connected Component (CC)

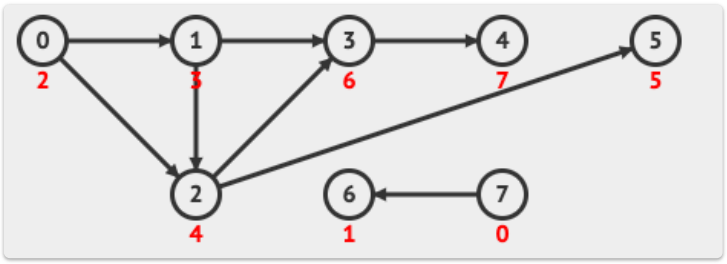
- Call `DFS(s)` or `BFS(s)` to enumerate all vertex v that has `status[v] = visited`

Counting the number of Connected Component

```
CC = 0
for all u in V:
        status[u] = unvisited
for all u in V:
        if (status[u] == unvisited):
                CC += 1
                DFS(u) # DFS will visit all nodes connected to this source node
print(CC)
```

Topological Sort

- DAG的拓扑排序是对DAG顶点的线性排序，其中每个顶点都排在他有出度的所有顶点之前
- 每个DAG有向无环图都至少有一个拓扑排序



8. Single-Source Shortest Path (SSSP)

- 在此问题中，我们需要在一个有向加权图中找到从特定的顶点到其他所有顶点的最短路径
- SSSP算法中的输入为：
  - 不一定相连的有向加权图G(V,E)
  - 源顶点s
- 输出为：
  - 大小为 V 的数组 D（D 代表 "距离）
    - 最初，如果 u = s，D[u] = 0；否则，D[u] = +∞
    - 当我们找到更好（更短）的路径时，D[u]就会减少
  - 大小为 V 的数组/矢量 p（p 代表 "父数组"/"前置数组"/"前一个数组"）
    - p[u] = 从源节点 s 到 u 的最佳路径上的前置因子
    - p[u] = NULL（未定义，我们可以使用-1 这样的值来表示）
    - 该数组/向量 p 描述了生成的 SSSP 生成树

8.1 Relax operation

```
def relax(u, v, w_u_v):
        if D[v] > D[u]+w_u_v: // if the path can be shortened
                D[v] = D[u]+w_u_v // 'relax' (update) the previous edge to shorter edge
                p[v] = u // remember/update the predecessor
```

- 当我们找到了一个更短的路径时，我们需要进行relax/update 操作，将现有的路径替换(update/relax) 成更短的路径

8.2 Approaches

- 我们可以对不同的图使用不同的算法来解决SSSP问题:
  - Unweighted Graphs 无权图: **BFS** $O(V+E)$
  - Graphs without negative weight 无负权图: **Dijkstra's Algorithm** $O((V+E)logV)$

- Graph without negative cycle 无负权循环图: **Modified Dijkstra's** $O((V+E)logV)$
- Tree 树: **DFS/BFS** $O(V+E)$
- Directed Acyclic Graphs (DAG)有向无环图: **Dynamic Programming** $O(V+E)$

Unweighted Graphs: BFS

在无权重图中，两个节点之间的"最短路径"简单地指的是连接它们的边数最少的路径。BFS 在这种情况下非常合适，因为它按层次（即边数）访问节点，从而能够保证找到的路径是边数最少的。
以下是使用 BFS 解决无权重图的 SSSP 问题的步骤：

1. **初始化**：首先，创建一个队列用于 BFS，并将起点（源点）入队。同时，创建一个数组或字典来存储每个节点到源点的最短距离，初始时除了源点（距离设为0）之外的所有节点距离都设为无穷大或未定义。
2. **进行 BFS**：当队列不为空时，重复以下步骤：
   - 从队列中弹出一个节点。
   - 检查该节点的每个邻居：
     - 如果邻居的最短距离尚未确定（即仍为无穷大或未定义），则更新其最短距离（设置为当前节点的最短距离加1），并将该邻居节点入队。
3. **终止**：当队列为空时，算法结束。此时，存储节点最短距离的数组或字典中的值即为从源点到每个节点的最短路径长度。

```
from collections import deque
def bfs_sssp(graph, start):
        distances = {vertex: float('infinity') for vertex in graph}
        distances[start] = 0

        queue = deque([start])

        while queue:
                current = queue.popleft()
                for neighbor in graph[current]:
                        if distances[neighbor] == float('infinity'):
                                distances[neighbor] = distances[current] + 1
                                queue.append(neighbor)

        return distances

# 示例图
graph = {
        'A': ['B', 'C'],
        'B': ['A', 'D', 'E'],
        'C': ['A', 'F'],
        'D': ['B'],
        'E': ['B', 'F'],
        'F': ['C', 'E']
}
# 计算从节点 'A' 到所有其他节点的最短距离
print(bfs_sssp(graph, 'A'))
```

Graphs without negative weight: Dijkstra's Algorithm

Dijkstra算法适用于有向和无向图，并假定所有边的权重都是非负的。它利用贪心策略，逐步确定从源点到图中所有其他顶点的最短路径。
Dijkstra算法的基本步骤:

1. **初始化**：对于图中的每个顶点，将其最短路径估计设置为无穷大，除了源点，其最短路径估计设为0。维护一个优先队列（最小堆），最初包含所有顶点，优先队列按照顶点的最短路径估计进行排序。
2. **处理顶点**：当优先队列非空时，重复以下步骤：
   - 从优先队列中取出最短路径估计最小的顶点（称为当前顶点）。
   - 对于当前顶点的每个相邻顶点，更新其最短路径估计。如果通过当前顶点到达邻居的路径比已知的路径更短，则更新邻居的最短路径估计，并在优先队列中更新其位置。
3. **算法结束**：当优先队列为空时，算法结束。此时，每个顶点的最短路径估计就是从源点到该顶点的最短路径。

```python
import heapq

def dijkstra(graph, start):
    # 初始化距离表
    distances = {vertex: float('infinity') for vertex in graph}
    distances[start] = 0
    # 初始化优先队列，并放入起点
    priority_queue = [(0, start)]

    while priority_queue:
        current_distance, current_vertex = heapq.heappop(priority_queue)
        # 如果该顶点的距离已经是最小了，则无需处理
        if current_distance > distances[current_vertex]:
            continue
        # 检查当前顶点的邻居
        for neighbor, weight in graph[current_vertex].items():
            distance = current_distance + weight
            # 更新邻居的距离
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(priority_queue, (distance, neighbor))

    return distances

# 示例图
graph = {
    'A': {'B': 1, 'C': 4},
    'B': {'A': 1, 'D': 2, 'E': 5},
    'C': {'A': 4, 'F': 5},
    'D': {'B': 2},
    'E': {'B': 5, 'F': 1},
    'F': {'C': 5, 'E': 1}
}
```

- 在此算法中，每个顶点从优先队列中提取一次$O(logV)$，由于有V个顶点，所以此步骤的运行速度为$O(VlogV)$
- 接下来检查当前顶点的邻居，更新邻居的距离需要用到relax操作，这部分的运行速度为$O(ElogV)$
- 总的运行速度为$O((E+V)logV)$

Trees: DFS/BFS

由于树没有循环，所以我们可以简单地从源点遍历树，同时累加沿路径的权重，以此计算到每个顶点的最短路径。

由于树是一种没有循环的特殊图，所以在树中，从一个顶点到另一个顶点的路径是唯一的。这意味着无论是使用 DFS 还是 BFS，当你首次到达一个顶点时，你找到的就是从源点到这个顶点的最短（也是唯一的）路径。

DFS

```python
def dfs(tree, node, current_distance, distances, visited):
    visited.add(node)
    distances[node] = current_distance

    for child, weight in tree[node].items():
        if child not in visited:
            dfs(tree, child, current_distance + weight, distances, visited)

def sssp_with_dfs(tree, start):
    distances = {}
    visited = set()
    dfs(tree, start, 0, distances, visited)
    return distances

# 示例树
tree = {
    'A': {'B': 1, 'C': 2},
    'B': {'A': 1, 'D': 4, 'E': 5},
    'C': {'A': 2, 'F': 6},
    'D': {'B': 4},
    'E': {'B': 5},
    'F': {'C': 6}
```

```
}

# 计算从节点 'A' 到所有其他节点的最短距离
print(sssp_with_dfs(tree, 'A'))
```

BFS

```
from collections import deque

def sssp_with_bfs(tree, start):
    distances = {start: 0}
    queue = deque([start])

    while queue:
        node = queue.popleft()
        for child, weight in tree[node].items():
            if child not in distances:
                distances[child] = distances[node] + weight
                queue.append(child)

    return distances

# 使用相同的树结构和起点调用 BFS
print(sssp_with_bfs(tree, 'A'))
```

在这两种方法中，我们都维护了一个 `distances` 字典来记录从源点到每个节点的最短距离。由于树中的路径是唯一的，所以当我们第一次访问每个节点时，计算的距离就是最短距离。DFS 和 BFS 在这种情况下都是有效的，选择哪种方法取决于个人偏好或特定应用场景的需求。

Directed Acyclic Graphs: Dynamic Programming

动态规划在这类问题中的关键在于利用 DAG 的拓扑排序，确保在计算一个顶点的最短路径时，已经计算了其所有前驱节点的最短路径。

动态规划解决 DAG 的 SSSP 问题的步骤如下：

1. **拓扑排序**：首先对 DAG 进行拓扑排序。拓扑排序是将图中的顶点排成一个线性序列，使得对于图中的每条边 `u -> v`，`u` 在序列中都出现在 `v` 之前。拓扑排序可以通过 DFS 实现。
2. **初始化距离**：将源点到所有顶点的距离初始化为无穷大，除了源点自身的距离为 0。
3. **动态规划更新**：按照拓扑排序的顺序，更新每个顶点的最短路径。对于每个顶点，检查所有进入该顶点的边，并更新该顶点的最短路径。

```
from collections import defaultdict, deque

# 拓扑排序的实现
def topological_sort(graph):
    def dfs(node):
        visited.add(node)
        for neighbor in graph[node]:
            if neighbor not in visited:
                dfs(neighbor)
        order.appendleft(node)

    visited = set()
    order = deque()
    for vertex in graph:
        if vertex not in visited:
            dfs(vertex)
    return order
# 使用 DP 解决 SSSP 问题
def sssp_in_dag(graph, start):
    # 拓扑排序
    topo_order = topological_sort(graph)

    # 初始化距离表
    distances = {vertex: float('infinity') for vertex in graph}
```

```
    distances[start] = 0

    # 遍历拓扑排序的顺序，更新距离
    for vertex in topo_order:
        for neighbor, weight in graph[vertex].items():
            distances[neighbor] = min(distances[neighbor], distances[vertex] + weight)

    return distances

# 示例 DAG
dag = {
    'A': {'B': 3, 'C': 6},
    'B': {'C': 4, 'D': 4, 'E': 11},
    'C': {'D': 8, 'G': 11},
    'D': {'E': -4, 'F': 5, 'G': 2},
    'E': {'H': 9},
    'F': {'H': 1},
    'G': {'H': 2},
    'H': {}
}
# 计算从节点 'A' 到所有其他节点的最短距离
print(sssp_in_dag(dag, 'A'))
```

在这个实现中，首先通过 `topological_sort` 函数得到 DAG 的拓扑排序。然后，在 `sssp_in_dag` 函数中使用这个排序来确保在计算每个顶点的最短路径时，其所有前驱节点的最短路径都已经计算过了。这样，我们可以保证动态规划的每一步都是基于正确和最新的信息。

9. Pastpaper related

9.1 Python implement ADT

- **Linked List, Doubly Linked List**: 自定义类来实现
- **Stack**: 使用list实现
- **Queue, Double-Ended Queue(Deque)**: 使用`collections.deque`实现，此库实现的是双端队列，通过禁用`appendleft`和`pop`来实现
- **Binary Heap**: 使用list实现，同时使用`heapq`模块实现堆操作。`heapq`实现的是最小堆操作，对于最大堆，需要将list中的元素取反
- **Priority Queue**: 使用Binary Heap实现
- **Binary Search Tree (BST)**: 自定义类实现
- **Adjacency Matrix**: 二维数组实现，`AM[i][j]`表示从节点i到节点j的边，所存储的值为此边的权重
- **Adjacency List**: 字典实现，键为顶点，值为邻接节点的列表
- **Edge List**: 列表实现，列表中的每一个元素为一个表示边的元组`set(node1, node2, weight)`

9.2 Time Complexity for algorithms and DS operations

Sorting

- **Comparison Based Sorting**
  - **Bubble Sort, Selection Sort, Insertion Sort** = $O(N^2)$
  - **Merge Sort, Quick Sort** = $O(NlogN)$
- **Non Comparison Based Sorting Algorithm**
  - **Counting Sort** = $O(N)$
    - 在对于最大和最小值之差不是很大，且重复值多的整数数组排序时非常高效
    - 然而对于非整数排序则需要调整
    - 如果数值范围很大，即使只有几个数，也需要大量的空间复杂度
  - **Radix Sort** = $O(nk)$, n是排序的项数，k是数字平均长度
    - 其需要更多的空间来存储桶
    - 只可用于分解为独立数位的数据类型（整数或长字符串）

Python Array (List)

- 索引 `list[i]`: 时间复杂度为O(1)。因为列表是通过连续内存空间存储元素，所以可以直接根据索引快速访问元素。
- 末尾添加元素（例如 `list.append(item)`） - 平均时间复杂度为O(1)。尽管偶尔需要扩展内存空间来容纳更多元素，导致时间复杂度增加，但通常情况下，添加元素是非常快的。
- 末尾删除元素（例如 `list.pop()`） - 时间复杂度为O(1)。由于删除的是最后一个元素，所以这个操作非常快速。
- 插入或删除指定位置的元素（例如 `list.insert(i, item)` 或 `list.pop(i)`） - 时间复杂度为O(n)。这是因为插入或删除元素需要移动该位置后面的所有元素来保持列表的连续性。
- 查找元素（例如 `item in list`） - 时间复杂度为O(n)。在最坏的情况下，可能需要遍历整个列表来查找元素。
- 列表长度（例如 `len(list)`） - 时间复杂度为O(1)。Python内部会存储列表的长度，因此获取长度是即时的。
- 列表排序（例如 `list.sort()` 或 `sorted(list)`） - 时间复杂度为O(n log n)。Python使用的排序算法是Timsort，这是一种高效的排序算法，适用于多种类型的数据。
- 列表切片（例如 `list[start:end]`） - 时间复杂度为O(k)，其中k是切片的长度。切片操作需要创建新列表并复制元素。
- 末尾添加元素（例如 `list.append(item)`） - 平均时间复杂度为O(1)。尽管偶尔需要扩展内存空间来容纳更多元素，导致时间复杂度增加，但通常情况下，添加元素是非常快的。
- 末尾删除元素（例如 `list.pop()`） - 时间复杂度为O(1)。由于删除的是最后一个元素，所以这个操作非常快速。
- 插入或删除指定位置的元素（例如 `list.insert(i, item)` 或 `list.pop(i)`） - 时间复杂度为O(n)。这是因为插入或删除元素需要移动该位置后面的所有元素来保持列表的连续性。
- 查找元素（例如 `item in list`） - 时间复杂度为O(n)。在最坏的情况下，可能需要遍历整个列表来查找元素。
- 列表长度（例如 `len(list)`） - 时间复杂度为O(1)。Python内部会存储列表的长度，因此获取长度是即时的。
- 列表排序（例如 `list.sort()` 或 `sorted(list)`） - 时间复杂度为O(n log n)。Python使用的排序算法是Timsort，这是一种高效的排序算法，适用于多种类型的数据。
- 列表切片（例如 `list[start:end]`） - 时间复杂度为O(k)，其中k是切片的长度。切片操作需要创建新列表并复制元素。

9.3 Graph

- **Subgraph 子图**: 子图中的所有顶点都在原图的顶点集中，子图的所有边都在原图的边集中
- **Simple path 简单图**：没有重边（两个顶点之间只有一条边）和自环（顶点到自身的边）
- **Connected Graph 连通图**: 每个节点可以通过一个或多个节点到达图中的任意一个节点
- **Connected Component (undirected graph only)无向图中的连通成分**: 在一个无向图中，一个连通成分是指图中的一个最大子图，其中任意两个顶点都通过图中的边相连。换句话说，在同一个连通成分中的任何两个顶点都至少存在一条路径相互到达。
  - 如果一个无向图的任意两个顶点都是连通的，那么这个图被称为连通图。
  - 如果无向图不是连通图，它可以被分解为若干个连通成分。
- **Strongly Connected Component 有向图中的强连通成分**：在一个有向图中，一个强连通成分是指图中的一个最大子图，其中任意两个顶点 u 和 v 都存在一条从 u 到 v 的有向路径，同时也存在一条从 v 到 u 的有向路径。
  - 如果一个有向图中的任意两个顶点都满足这种双向可达的条件，那么这个图被称为强连通图。
  - 在大多数有向图中，可以找到多个强连通成分，每个成分内的顶点都是相互强连通的。
- **Trivial Cycle 平凡环**: 仅包含单个顶点的环。换句话说，平凡环是图中一个顶点的最简单形式的环。在讨论图中的环结构时，通常环是指由至少三个顶点组成的闭合路径，这些顶点通过边相互连接，且不重复访问任何顶点。

Tree

树是一个连通图，具有 V 个顶点和 $E = V - 1$ 条边
树的特点:

- 无环：不包含任何环
- 任意两个顶点之间有*唯一的路径*
- 树通常定义在无向图上
- 将树中的一个顶点指定为*根顶点*后，称之为有根树
- 在有根树中，我们有层级结构（父节点、子节点、祖先、后代）的概念，以及子树、层级和高度。
  - **Parent 父节点**
    - 每个节点（除了根节点）都有一个直接相连的上级节点称为其父节点

- **Children 子节点**
  - 任何节点的直接下级节点称为它的子节点
- **Ancestors 祖先**
  - 从根节点到达某个节点所经过的所有节点，**包括该节点本身**，都是这个节点的祖先
- **Descendants 后代**
  - 任何节点下方的所有节点（无论距离多远）都是该节点的后代
- **Subtrees 子树**
  - 任何节点**和**它的后代节点，连同它们之间的边，一起构成一个子树
- **Levels 层**
  - 根节点位于第一层，其子节点位于第二层，以此类推。某个节点的层级数是从根节点到该节点的唯一路径上的边的数量加一
- **Height 高**
  - 树的高度是树中任何节点的最大层级数。等价地，也可以定义为从该节点到其任何叶节点的最长路径的长度

- **二叉树 Binary Tree** 是一种 *有根树* rooted tree，其中每个顶点最多有两个子节点，分别称为left child和right child
- **满二叉树 Full Binary Tree**是一种二叉树，其中每个非叶子节点（也称为内部节点）*恰好有两个子节点*
- **完全二叉树 Complete Binary Tree**是一种二叉树，其中每一层都完全填满，除了可能最后一层可能未完全填满，但最后一层的填充应尽可能地*向左*进行。

Complete Graph

- **完全图 Complete Graph**是一种图，其中任意两个顶点之间都存在一条边
- 完全图是最密集的简单图

Bipartite Graph

- **二分图 Bipartite Graph**是一种无向图，具有 V 个顶点，这些顶点可以被划分为两个大小分别为 m 和 n 的相邻顶点集合，其中 $V = m + n$。
- 同一集合中的顶点之间没有边。
- 二分图不包含奇数长度的环
- 二分图可以是*完全图 Complete Graph*，即一个集合中的所有 m 个顶点都与另一个集合中的所有 n 个顶点相连
- **树也是一种二分图。**

9.4 SSSP

- 原始的Dijkstra算法无法高效的用python实现，因为`heapq`库中的decreaseKey操作具有lazy manner特性
- 对于修改过的Dijkstra算法，有可能陷入无限循环，即当源节点可以遍历道一个有负权重的环，因为Dijkstra算法是greedy的，所以有负权重环时不会进一步传播
- 我们可以对不同的图使用不同的算法来解决SSSP问题:
  - Unweighted Graphs 无权图: **BFS** $O(V + E)$
  - Graphs without negative weight 无负权图: **Dijkstra's Algorithm** $O((V + E)logV)$
  - Graph without negative cycle 无负权循环图: **Modified Dijkstra's** $O((V + E)logV)$
  - Tree 树: **DFS/BFS** $O(V + E)$
  - Directed Acyclic Graphs (DAG)有向无环图: **Dynamic Programming** $O(V + E)$
- Unweighted Graph可以被Dijkstra和modified Dijsktra解决，但是BFS更高效