Cheeting Sheet

一、图

1.BFS

- BFS 适合于解决需要找到最短路径的问题，因为它会逐层地搜索所有可能的路径，从起点开始，一层一层地向外扩展。在解决迷宫问题时，BFS 可以找到最短路径（如果存在），但相对于 DFS 来说，它可能需要更多的空间来存储搜索过程中的状态信息。
- BFS 的优点是能够找到最短路径，并且保证在找到解之前不会漏掉任何可能的路径。

```python
from collections import defaultdict, deque

# Class to represent a graph using adjacency list
class Graph:
    def __init__(self):
        self.adjList = defaultdict(list)

    # Function to add an edge to the graph
    def addEdge(self, u, v):
        self.adjList[u].append(v)

    # Function to perform Breadth First Search on a graph represented using
adjacency list
    def bfs(self, startNode):
        # Create a queue for BFS
        queue = deque()
        visited = set()

        # Mark the current node as visited and enqueue it
        visited.add(startNode)
        queue.append(startNode)

        # Iterate over the queue
        while queue:
            # Dequeue a vertex from queue and print it
            currentNode = queue.popleft()
            print(currentNode, end=" ")

            # Get all adjacent vertices of the dequeued vertex currentNode
            # If an adjacent has not been visited, then mark it visited and
enqueue it
            for neighbor in self.adjList[currentNode]:
                if neighbor not in visited:
                    visited.add(neighbor)
                    queue.append(neighbor)
```

2.DFS

- DFS 适合于解决能够表示成树形结构的问题，包括迷宫问题。DFS 会尽可能地沿着迷宫的一条路径向前探索，直到不能再前进为止，然后回溯到前一个位置，再尝试其他路径。在解决迷宫问题时，DFS 可以帮助我们快速地找到一条通路（如果存在），但不一定能够找到最短路径。
- DFS 的优点是实现简单，不需要额外的数据结构来保存搜索状态。

```python
from collections import defaultdict


class Graph:
    def __init__(self):
        self.graph = defaultdict(list)

    def addEdge(self, u, v):
        self.graph[u].append(v)

    def DFS(self, v, visited=None):
        if visited is None:
            visited = set()
        visited.add(v)
        print(v, end=' ')
        for neighbour in self.graph[v]:
            if neighbour not in visited:
                self.DFS(neighbour, visited)
```

3.最短路径算法

1)Dijkstra

用于找到两个顶点之间的最短路径

```python
from pythonds3.trees.binary_heap import BinaryHeap
class PriorityQueue(BinaryHeap):
    def change_priority(self, search_key: Any, new_priority: Any) -> None:
        key_to_move = -1
        for i, (_, key) in enumerate(self._heap):
            if key == search_key:
                key_to_move = i
                break
        if key_to_move > -1:
            self._heap[key_to_move] = (new_priority, search_key)
            self._perc_up(key_to_move)

    def __contains__(self, search_key: Any) -> bool:
        for _, key in self._heap:
            if key == search_key:
                return True
        return False
```

```
from pythonds3.graphs import PriorityQueue
def dijkstra(graph,start):
    pq = PriorityQueue()
    start.setDistance(0)
    pq.buildHeap([(v.getDistance(),v) for v in graph])
    while pq:
        distance, current_v = pq.delete()
        for next_v in current_v.getneighbors():
            new_distance = current_v.distance + current_v.get_neighbor(next_v) #
+ get_weight
            if new_distance < next_v.distance:
                next_v.distance = new_distance
                next_v.previous = current_v
                pq.change_priority(next_v,new_distance)
```

2)Bellman-Ford

用于处理带有负权边的图的最短路径问题

3)Floyd-Warshall

用于找到图中所有顶点之间的最短路径

```
def floyd_warshall(graph):
    n = len(graph)
    dist = [[float('inf')] * n for _ in range(n)]

    for i in range(n):
        for j in range(n):
            if i == j:
                dist[i][j] = 0
            elif j in graph[i]:
                dist[i][j] = graph[i][j]

    for k in range(n):
        for i in range(n):
            for j in range(n):
                dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])

    return dist
```

4.最小生成树算法

1)Prim

用于找到连接所有顶点的最小生成树

```
from pythonds3.graphs import PriorityQueue

def prim(graph,start):
    pq = PriorityQueue()
    for vertex in graph:
        vertex.distance = sys.maxsize
        vertex.previous = None
    start.distance = 0
    pq.buildHeap([(v.distance,v) for v in graph])
    while pq:
        distance, current_v = pq.delete()
        for next_v in current_v.get_eighbors():
          new_distance = current_v.get_neighbor(next_v)
          if next_v in pq and new_distance < next_v.distance:
              next_v.previous = current_v
              next_v.distance = new_distance
              pq.change_priority(next_v,new_distance)
```

2)Krustal/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

```
class DisjointSet:
    def __init__(self, num_vertices):
        self.parent = list(range(num_vertices))
        self.rank = [0] * num_vertices

    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    def union(self, x, y):
        root_x = self.find(x)
        root_y = self.find(y)

        if root_x != root_y:
            if self.rank[root_x] < self.rank[root_y]:
                self.parent[root_x] = root_y
            elif self.rank[root_x] > self.rank[root_y]:
                self.parent[root_y] = root_x
            else:
                self.parent[root_x] = root_y
                self.rank[root_y] += 1


def kruskal(graph):
    num_vertices = len(graph)
    edges = []

    # 构建边集
    for i in range(num_vertices):
        for j in range(i + 1, num_vertices):
            if graph[i][j] != 0:
                edges.append((i, j, graph[i][j]))

    # 按照权重排序
    edges.sort(key=lambda x: x[2])

    # 初始化并查集
    disjoint_set = DisjointSet(num_vertices)
    # 构建最小生成树的边集
    minimum_spanning_tree = []

    for edge in edges:
        u, v, weight = edge
        if disjoint_set.find(u) != disjoint_set.find(v):
            disjoint_set.union(u, v)
            minimum_spanning_tree.append((u, v, weight))

    return minimum_spanning_tree
```

5.拓扑排序算法

1)DFS

用于对有向无环图（DAG）进行拓扑排序

2)Kahn/BFS

用于对有向无环图进行拓扑排序

```
from collections import deque, defaultdict

def topological_sort(graph):
    indegree = defaultdict(int)
    result = []
    queue = deque()

    # 计算每个顶点的入度
    for u in graph:
        for v in graph[u]:
            indegree[v] += 1

    # 将入度为 0 的顶点加入队列
    for u in graph:
        if indegree[u] == 0:
            queue.append(u)

    # 执行拓扑排序
    while queue:
        u = queue.popleft()
        result.append(u)

        for v in graph[u]:
            indegree[v] -= 1
            if indegree[v] == 0:
                queue.append(v)

    # 检查是否存在环
    if len(result) == len(graph):
        return result
    else:
        return None

# 示例调用代码
```

```python
graph = {
    'A': ['B', 'C'],
    'B': ['C', 'D'],
    'C': ['E'],
    'D': ['F'],
    'E': ['F'],
    'F': []
}

sorted_vertices = topological_sort(graph)
if sorted_vertices:
    print("Topological sort order:", sorted_vertices)
else:
    print("The graph contains a cycle.")
```

6.强连通分量算法

1)Kosaraju/2DFS

用于找到有向图中的所有强连通分量

```python
def dfs1(graph, node, visited, stack):
    visited[node] = True
    for neighbor in graph[node]:
        if not visited[neighbor]:
            dfs1(graph, neighbor, visited, stack)
    stack.append(node)

def dfs2(graph, node, visited, component):
    visited[node] = True
    component.append(node)
    for neighbor in graph[node]:
        if not visited[neighbor]:
            dfs2(graph, neighbor, visited, component)

def kosaraju(graph):
    # Step 1: Perform first DFS to get finishing times
    stack = []
    visited = [False] * len(graph)
    for node in range(len(graph)):
        if not visited[node]:
            dfs1(graph, node, visited, stack)

    # Step 2: Transpose the graph
    transposed_graph = [[] for _ in range(len(graph))]
    for node in range(len(graph)):
        for neighbor in graph[node]:
            transposed_graph[neighbor].append(node)

    # Step 3: Perform second DFS on the transposed graph to find SCCs
    visited = [False] * len(graph)
    sccs = []
    while stack:
        node = stack.pop()
        if not visited[node]:
            scc = []
            dfs2(transposed_graph, node, visited, scc)
            sccs.append(scc)
    return sccs

# Example
graph = [[1], [2, 4], [3, 5], [0, 6], [5], [4], [7], [5, 6]]
sccs = kosaraju(graph)
print("Strongly Connected Components:")
for scc in sccs:
    print(scc)
```

2)Tarjan

用于找到有向图中的所有强连通分量

```python
def tarjan(graph):
    def dfs(node):
        nonlocal index, stack, indices, low_link, on_stack, sccs
        index += 1
        indices[node] = index
        low_link[node] = index
        stack.append(node)
        on_stack[node] = True

        for neighbor in graph[node]:
            if indices[neighbor] == 0:  # Neighbor not visited yet
                dfs(neighbor)
                low_link[node] = min(low_link[node], low_link[neighbor])
            elif on_stack[neighbor]:  # Neighbor is in the current SCC
                low_link[node] = min(low_link[node], indices[neighbor])

        if indices[node] == low_link[node]:
            scc = []
            while True:
                top = stack.pop()
                on_stack[top] = False
                scc.append(top)
                if top == node:
                    break
            sccs.append(scc)

    index = 0
    stack = []
    indices = [0] * len(graph)
    low_link = [0] * len(graph)
    on_stack = [False] * len(graph)
    sccs = []

    for node in range(len(graph)):
        if indices[node] == 0:
            dfs(node)

    return sccs
```

```python
# Example
graph = [[1],          # Node 0 points to Node 1
         [2, 4],       # Node 1 points to Node 2 and Node 4
         [3, 5],       # Node 2 points to Node 3 and Node 5
         [0, 6],       # Node 3 points to Node 0 and Node 6
         [5],          # Node 4 points to Node 5
         [4],          # Node 5 points to Node 4
         [7],          # Node 6 points to Node 7
         [5, 6]]       # Node 7 points to Node 5 and Node 6

sccs = tarjan(graph)
print("Strongly Connected Components:")
for scc in sccs:
    print(scc)
```

二、树

1.并查集

```python
class DisjSet:
    def __init__(self, n):
        f.rank = [1] * n
        self.parent = [i for i in range(n)]

    def find(self, x):
        if (self.parent[x] != x): self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    def Union(self, x, y):
        xset, yset = self.find(x), self.find(y)
        if xset == yset: return

        if self.rank[xset] < self.rank[yset]: self.parent[xset] = yset
        elif self.rank[xset] > self.rank[yset]: self.parent[yset] = xset
        else: self.parent[yset] = xset; self.rank[xset] = self.rank[xset] + 1
```

2.二叉树层次遍历

```python
from collections import deque
def level_order_traversal(root):
    if root is None:
        return []
    result = []
    queue = deque([root])
    while queue:
        node = queue.popleft()
        result.append(node.data)
        if node.left:
            queue.append(node.left)
        if node.right:
            queue.append(node.right)
    return result
```

3.22161:哈夫曼编码树

```python
import heapq

class Node:
    def __init__(self, weight, char=None):
        self.weight = weight
        self.char = char
        self.left = None
        self.right = None

    def __lt__(self, other):
        if self.weight == other.weight:
            return self.char < other.char
        return self.weight < other.weight

def build_huffman_tree(characters):
    heap = []
    for char, weight in characters.items():
        heapq.heappush(heap, Node(weight, char))

    while len(heap) > 1:
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)
        #merged = Node(left.weight + right.weight) #note: 合并后，char 字段默认值是空
        merged = Node(left.weight + right.weight, min(left.char, right.char))
        merged.left = left
        merged.right = right
        heapq.heappush(heap, merged)

    return heap[0]

def encode_huffman_tree(root):
    codes = {}

    def traverse(node, code):
        #if node.char:
        if node.left is None and node.right is None:
            codes[node.char] = code
        else:
            traverse(node.left, code + '0')
            traverse(node.right, code + '1')

    traverse(root, '')
    return codes

def huffman_encoding(codes, string):
    encoded = ''
    for char in string:
        encoded += codes[char]
    return encoded

def huffman_decoding(root, encoded_string):
    decoded = ''
    node = root
    for bit in encoded_string:
        if bit == '0':
            node = node.left
        else:
            node = node.right

        #if node.char:
        if node.left is None and node.right is None:
            decoded += node.char
            node = root
    return decoded

# 读取输入
n = int(input())
characters = {}
for _ in range(n):
    char, weight = input().split()
    characters[char] = int(weight)

#string = input().strip()
#encoded_string = input().strip()

# 构建哈夫曼编码树
huffman_tree = build_huffman_tree(characters)

# 编码和解码
codes = encode_huffman_tree(huffman_tree)

strings = []
while True:
    try:
        line = input()
        strings.append(line)

    except EOFError:
        break

results = []
#print(strings)
for string in strings:
    if string[0] in ('0','1'):
        results.append(huffman_decoding(huffman_tree, string))
    else:
        results.append(huffman_encoding(codes, string))

for result in results:
    print(result)
```

4.二叉搜索树实现快排

```
class TreeNode:
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None
```

```
def insert(root, val):
    if root is None:
        return TreeNode(val)
    if val < root.val:
        root.left = insert(root.left, val)
    else:
        root.right = insert(root.right, val)
    return root

def inorder_traversal(root, result):
    if root:
        inorder_traversal(root.left, result)
        result.append(root.val)
        inorder_traversal(root.right, result)

def quicksort(nums):
    if not nums:
        return []
    root = TreeNode(nums[0])
    for num in nums[1:]:
        insert(root, num)
    result = []
    inorder_traversal(root, result)
    return result
```

5.AVL树

```
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None
        self.height = 1

class AVL:
    def __init__(self):
        self.root = None

    def insert(self, value):
        if not self.root:
            self.root = Node(value)
        else:
            self.root = self._insert(value, self.root)

    def _insert(self, value, node):
        if not node:
            return Node(value)
        elif value < node.value:
            node.left = self._insert(value, node.left)
        else:
            node.right = self._insert(value, node.right)

        node.height = 1 + max(self._get_height(node.left),
self._get_height(node.right))
```

```
        balance = self._get_balance(node)

        if balance > 1:
            if value < node.left.value: # 树形是 LL
                return self._rotate_right(node)
            else:   # 树形是 LR
                node.left = self._rotate_left(node.left)
                return self._rotate_right(node)

        if balance < -1:
            if value > node.right.value:    # 树形是 RR
                return self._rotate_left(node)
            else:   # 树形是 RL
                node.right = self._rotate_right(node.right)
                return self._rotate_left(node)

        return node

    def _get_height(self, node):
        if not node:
            return 0
        return node.height

    def _get_balance(self, node):
        if not node:
            return 0
        return self._get_height(node.left) - self._get_height(node.right)

    def _rotate_left(self, z):
        y = z.right
        T2 = y.left
        y.left = z
        z.right = T2
        z.height = 1 + max(self._get_height(z.left), self._get_height(z.right))
        y.height = 1 + max(self._get_height(y.left), self._get_height(y.right))
        return y

    def _rotate_right(self, y):
        x = y.left
        T2 = x.right
        x.right = y
        y.left = T2
        y.height = 1 + max(self._get_height(y.left), self._get_height(y.right))
        x.height = 1 + max(self._get_height(x.left), self._get_height(x.right))
        return x

    def preorder(self):
        return self._preorder(self.root)

    def _preorder(self, node):
        if not node:
            return []
        return [node.value] + self._preorder(node.left) +
self._preorder(node.right)
```

```
n = int(input().strip())
sequence = list(map(int, input().strip().split()))

avl = AVL()
for value in sequence:
    avl.insert(value)

print(' '.join(map(str, avl.preorder())))
```

```
class AVL:
    # Existing code...

    def delete(self, value):
        self.root = self._delete(value, self.root)

    def _delete(self, value, node):
        if not node:
            return node

        if value < node.value:
            node.left = self._delete(value, node.left)
        elif value > node.value:
            node.right = self._delete(value, node.right)
        else:
            if not node.left:
                temp = node.right
                node = None
                return temp
            elif not node.right:
                temp = node.left
                node = None
                return temp

            temp = self._min_value_node(node.right)
            node.value = temp.value
            node.right = self._delete(temp.value, node.right)

        if not node:
            return node

        node.height = 1 + max(self._get_height(node.left),
self._get_height(node.right))

        balance = self._get_balance(node)

        # Rebalance the tree
        if balance > 1:
            if self._get_balance(node.left) >= 0:
                return self._rotate_right(node)
            else:
                node.left = self._rotate_left(node.left)
                return self._rotate_right(node)

        if balance < -1:
            if self._get_balance(node.right) <= 0:
                return self._rotate_left(node)
            else:
                node.right = self._rotate_right(node.right)
                return self._rotate_left(node)

        return node

    def _min_value_node(self, node):
        current = node
        while current.left:
            current = current.left
        return current

    # Existing code...
```

```
else:
    if not node.left:
        temp = node.right
        node = None
        return temp
    elif not node.right:
        temp = node.left
        node = None
        return temp

    temp = self._min_value_node(node.right)
    node.value = temp.value
    node.right = self._delete(temp.value, node.right)
```

三、其他

1.02754八皇后

```
def queen_stack(n):
    stack = []  # 用于保存状态的栈
    solutions = [] # 存储所有解决方案的列表

    stack.append((0, []))  # 初始状态为第一行，所有列都未放置皇后,栈中的元素是 (row, queens) 的元组

    while stack:
        row, cols = stack.pop() # 从栈中取出当前处理的行数和已放置的皇后位置
        if row == n:     # 找到一个合法解决方案
            solutions.append(cols)
        else:
            for col in range(n):
                if is_valid(row, col, cols): # 检查当前位置是否合法
                    stack.append((row + 1, cols + [col]))

    return solutions

def is_valid(row, col, queens):
    for r in range(row):
        if queens[r] == col or abs(row - r) == abs(col - queens[r]):
            return False
    return True


# 获取第 b 个皇后串
def get_queen_string(b):
    solutions = queen_stack(8)
    if b > len(solutions):
        return None
    b = len(solutions) + 1 - b

    queen_string = ''.join(str(col + 1) for col in solutions[b - 1])
    return queen_string

test_cases = int(input())  # 输入的测试数据组数
for _ in range(test_cases):
    b = int(input())  # 输入的 b 值
    queen_string = get_queen_string(b)
    print(queen_string)
```

2.链表

1)单向链表

```
class LinkList:
    class Node:
        def __init__(self, data, next=None):
            self.data, self.next = data, next

    def __init__(self):
        self.head = self.tail = LinkList.Node(None, None)
        self.size = 0

    def print(self):
        ptr = self.head
        while ptr is not None:
            print(ptr.data, end=',')
            ptr = ptr.next

    def insert(self, p, data):
        nd = LinkList.Node(data, None)
        if self.tail is p:
            self.tail = nd
        nd.next = p.next
        p.next = nd
        self.size += 1

    def delete(self, p):
        if self.tail is p.next:
            self.tail = p
        p.next = p.next.next
        self.size -= 1

    def popFront(self):
        if self.head is None:
            raise Exception("Popping front for Empty link list.")
        else:
            self.head = self.head.next
            self.size -= 1
            if self.size == 0:
                self.head = self.tail = None

    def pushFront(self, data):
        nd = LinkList.Node(data, self.head)
        self.head = nd
        self.size += 1
        if self.size == 1:
            self.tail = nd

    def pushBack(self, data):
        if self.size == 0:
            self.pushFront(data)
        else:
            self.insert(self.tail, data)

    def clear(self):
        self.head = self.tail = None
        self.size = 0

    def __iter__(self):
        self.ptr = self.head
        return self

    def __next__(self):
        if self.ptr is None:
            raise StopIteration()
        else:
            data = self.ptr.data
            self.ptr = self.ptr.next
            return data
```

2)双向链表

```
class DoubleLinkList:
    class Node:
        def __init__(self, data, prev=None, next=None):
            self.data, self.prev, self.next = data, prev, next

    class Iterator:
        def __init__(self, p):
            self.ptr = p

        def get(self):
            return self.ptr.data

        def set(self, data):
            self.ptr.data = data

        def __iter__(self):
            self.ptr = self.ptr.next
```

```python
            if self.ptr is None:
                return None
            else:
                return DoubleLinkList.Iterator(self.ptr)

        def prev(self):
            self.ptr = self.ptr.prev
            return DoubleLinkList.Iterator(self.ptr)

    def __init__(self):
        self.head = self.tail = DoubleLinkList.Node(None, None, None)
        self.size = 0

    def _insert(self, p, data):
        nd = DoubleLinkList.Node(data, p, p.next)
        if self.tail is p:
            self.tail = nd
        if p.next:
            p.next.prev = nd
        p.next = nd
        self.size += 1

    def _delete(self, p):
        if self.size == 0 or p is self.head:
            return Exception("Illegal deleting.")
        else:
            p.prev.next = p.next
            if p.next:
                p.next.prev = p.prev
            if self.tail is p:
                self.tail = p.prev
            self.size -= 1

    def clear(self):
        self.tail = self.head
        self.head.next = self.head.prev = None
        self.size = 0

    def begin(self):
        return DoubleLinkList.Iterator(self.head.next)

    def end(self):
        return None

    def insert(self, i, data):
        self._insert(i.ptr, data)

    def delete(self, i):
        self._delete(i.ptr)

    def pushFront(self, data):
        self._insert(self.head, data)

    def pushBack(self, data):
        self._insert(self.tail, data)
    def popFront(self):
        self._delete(self.head.next)

    def popBack(self):
        self._delete(self.tail)

    def __iter__(self):
        self.ptr = self.head.next
        return self

    def __next__(self):
        if self.ptr is None:
            raise StopIteration()
        else:
            data = self.ptr.data
            self.ptr = self.ptr.next
            return data

    def find(self, val):
        ptr = self.head.next
        while ptr is not None:
            if ptr.data == val:
                return DoubleLinkList.Iterator(ptr)
            ptr = ptr.next
        return self.end()

    def printList(self):
        ptr = self.head.next
        while ptr is not None:
            print(ptr.data, end=',')
            ptr = ptr.next
```

3)循环链表

```
class CircleLinkList:
    class Node:
        def __init__(self, data, next=None):
            self.data, self.next = data, next

    def __init__(self):
        self.tail = None
        self.size = 0

    def is_empty(self):
        return self.size == 0

    def pushFront(self, data):
        nd = CircleLinkList.Node(data)
        if self.is_empty():
            self.tail = nd
            nd.next = self.tail
        else:
            nd.next = self.tail.next
            self.tail.next = nd
        self.size += 1
    def pushBack(self, data):
        self.pushFront(data)
        self.tail = self.tail.next

    def popFront(self):
        if self.is_empty():
            return None
        else:
            nd = self.tail.next
            self.size -= 1
            if self.size == 0:
                self.tail = None
            else:
                self.tail.next = nd.next
        return nd.data

    def popBack(self):
        if self.is_empty():
            return None
        else:
            nd = self.tail.next
            while nd.next != self.tail:
                nd = nd.next
            data = self.tail.data
            nd.next = self.tail.next
            self.tail = nd
            return data

    def printList(self):
        if self.is_empty():
            print('Empty!')
        else:
            ptr = self.tail.next
            while True:
                print(ptr.data, end=',')
                if ptr == self.tail:
                    break
                ptr = ptr.next
            print()
```

3.质数

```
def judge(number):
    nlist = list(range(1,number+1))
    nlist[0] = 0
    k = 2
    while k * k <= number:
        if nlist[k-1] != 0:
            for i in range(2*k,number+1,k):
                nlist[i-1] = 0
        k += 1
    result = []
    for num in nlist:
        if num != 0:
            result.append(num)
    return result
```