Cheetsheet for the final exam

语法

for _ in sorted(dic.keys()): ##将字典按keys排序后 float('inf') 表示正无穷 print(*lst) ##把列表中元素顺序输出 for key,value in dict.items() 遍历字典的键值对 list(zip(a,b)) 将两个列表元素一一配对，生成元组的列表。 str.strip(): 移除字符串空白字符。 保留n位小数：r'%.nf'%原数字；'{:.nf}'.format(原数字)； n位有效数字：'%.ng'%原数字；'{:.ng}'.format(原数字) str.isalpha() / str.isdigit() / str.isalnum(): 检查字符串是否全部由字母/数字/字母和数字组成。

二分查找

```
import bisect
sorted_list = [1,3,5,7,9] #[(0)1, (1)3, (2)5, (3)7, (4)9]
position = bisect.bisect_left(sorted_list, 6)#查找某一元素插入后的索引
print(position) # 输出：3，因为6应该插入到位置3，才能保持列表的升序顺序
bisect.insort_left(sorted_list, 6)#将某个元素插入原列表并不改变升序/降序
print(sorted_list) # 输出：[1, 3, 5, 6, 7, 9]，6被插入到适当的位置以保持升序顺序
sorted_list=(1,3,5,7,7,7,9)
print(bisect.bisect_left(sorted_list,7))
print(bisect.bisect_right(sorted_list,7))
# 输出：3 6
#右侧插入，如果有相同元素，就输出最大的索引，左侧输入则相反
```

中序表达式转后序表达式 Shunting Yard

```
def infix_to_postfix(expression):
    precedence = {'+':1, '-':1, '*':2, '/':2}
    stack = []
    postfix = []
    number = ''
    for char in expression:
        if char.isnumeric() or char == '.':
            number += char
        else:
            if number:
                num = float(number)
                postfix.append(int(num) if num.is_integer() else num)
                number = ''
            if char in '+-*/':
                while stack and stack[-1] in '+-*/' and precedence[char] <=
precedence[stack[-1]]:
                    postfix.append(stack.pop())
                stack.append(char)
            elif char == '(':
                stack.append(char)
            elif char == ')':
```

```
                while stack and stack[-1] != '(':
                    postfix.append(stack.pop())
                stack.pop()
        if number:
            num = float(number)
            postfix.append(int(num) if num.is_integer() else num)
        while stack:
            postfix.append(stack.pop())
        return ' '.join(str(x) for x in postfix)
    n = int(input())
    for _ in range(n):
        expression = input()
        print(infix_to_postfix(expression))
```

```
def judge(number):#埃氏筛法
    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
import math
def prime(n):#欧拉筛
    pr,l = [], [True] * (n+1)
    for i in range(2,n+1):
        if l[i]:
            pr.append(i)
        for j in pr:
            if i*j>n:
                break
            l[i*j]=False
            if i%j==0:
                break
    return pr,l
```

eval

eval() 将字符串当成有效的表达式来求值，并返回计算结果 eval 会把里面的字符串参数的引号去掉，把中间的内容当成Python的代码，eval 函数会执行这段代码并且返回执行结果 也可以这样来理解：eval() 函数就是实现 list、dict、tuple、与str 之间的转化

```

```

```
result = eval("1 + 1")
print(result)  # 2
result = eval("'+' * 5")
print(result)  # ++++
input_number = input("请输入一个加减乘除运算公式：")
print(eval(input_number))## 1*2 +3 ## 5
```

波兰表达式

波兰表达式是一种把运算符前置的算术表达式(2 + 3) * 4 *+234

```
s = input().split()
def cal():
    cur = s.pop(0)
    if cur in "+-*/":
        return str(eval(cal() + cur + cal()))
    else:
        return cur
print("%.6f" % float(cal()))
```

dp 最大上升子序列

```
input()
b = [int(x) for x in input().split()]
n = len(b)
dp = [0]*n
for i in range(n):
    dp[i] = b[i]
    for j in range(i):
        if b[j]<b[i]:
            dp[i] = max(dp[j]+b[i], dp[i])
print(max(dp))
```

递归

必须有一个明确的结束条件，每次进入更深一层递归时，问题规模（计算量）相比上次递归都应有所减少。 递归效率不高，递归层次过多会导致栈溢出（在计算机中，函数调用是通过栈（stack）这种数据结构实现的，每当进入一个函数调用，栈就会加一层栈帧，每当函数返回，栈就会减一层栈帧。由于栈的大小不是无限的，所以，递归调用的次数过多，会导致栈溢出

回溯递归（八皇后问题）

```
def backtrack(row):
    if row == n:  # 找到一个合法解决方案
        solutions.append(queens.copy())
```

```
    else:
        for col in range(n):
            if is_valid(row, col):  # 检查当前位置是否合法
                queens[row] = col  # 在当前行放置皇后
                backtrack(row + 1)  # 递归处理下一行
                queens[row] = -1  # 回溯，撤销当前行的选择
def is_valid(row, col):
    for r in range(row):
        if queens[r] == col or abs(row - r) == abs(col - queens[r]):
            return False
    return True
    backtrack(0)  # 从第一行开始回溯
    return solutions
def get_queen_string(b):
    solutions = solve_n_queens(8)
    if b > len(solutions):
        return None
    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)
```

树

1.树的建立

class法

```
class TreeNode:
    def __init__(self, value):
        # 二叉树 (binary tree)
        self.value = value
        self.left = None
        self.right = None
        # 多叉树 (N-nary tree)
        self.value = value
        self.children = []
        # 左儿子右兄弟树 (First child / Next sibling representation)
    self.value = value
    self.firstChild = None
    self.nextSibling = None
    # 这玩意像个链表，有时候会很好用
n = int(input())
# 一般而言会有一个存Nodes的dict或是list
nodes = [TreeNode() for i in range(n)]
# 甚至会让你找root，这也可以用于记录森林的树量
has_parents = [False] * n
for i in range(n):
```

```
    opt = map(int, input().spilt())
    if opt[0] != -1:
        nodes[i].left = nodes[opt[0]]
        has_parent[opt[0]] = True
    if opt[1] != -1:
        nodes[i].right = nodes[opt[1]]
        has_parent[opt[1]] = True
# 这里完成了树的建立
root = has_parent.index(False) # 对于一棵树而言，root可以被方便的确定
```

dict法

```
# 字典存储value和children，模拟TreeNode的存储结构
def parse_tree(s):
    stack = []
    node = None
    for char in s:
        if char.isalpha():  # 如果是字母，创建新节点
            node = {'value': char, 'children': []}
            if stack:  # 如果栈不为空，把节点作为子节点加入到栈顶节点的子节点列表中
                stack[-1]['children'].append(node)
        elif char == '(':  # 遇到左括号，当前节点可能会有子节点
            if node:
                stack.append(node)  # 把当前节点推入栈中
                node = None
        elif char == ')':  # 遇到右括号，子节点列表结束
            if stack:
                node = stack.pop()  # 弹出当前节点
    return node  # 根节点
```

```
#高度&深度
def tree_depth(node):
    if node is None:
        return 0
    left_depth = tree_depth(node.left)
    right_depth = tree_depth(node.right)
    return max(left_depth, right_depth) + 1 # 可以被修改为多叉树
#叶子数量
def count_leaves(node):
    if node is None:
        return 0
    if node.left is None and node.right is None:
        return 1
    return count_leaves(node.left) + count_leaves(node.right)
```

2. 遍历问题

前序遍历 在前序遍历中，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** 在中序遍历中，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** 在后序遍历中，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。

```
def pre_Order(root):
    if root is None:
        return []
    return root.value + pre_Order(root.left) + pre_Order(root.right)
def mid_Order(root):
    if root is None:
        return []
    return mid_Order(root.left) + root.value + mid_Order(root.right)
def post_Order(root):
    if root is None:
        return []
    return post_Order(root.left) + post_Order(root.right) + root.value
```

层级遍历（Level Order Traversal）# 利用BFS（deque）

```
import deque
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.children = []
def level_Order(root):
    queue = deque()
    queue.append(root)
    while (len(queue) != 0): #注意这是一个特殊的BFS,以层为单位
        n = len(queue)
        while (n > 0): #一层层的输出结果
            point = queue.popleft()
            print(point.value, end=" ") # 这里的输出是一行
            queue.extend(point.children)
            n -= 1
        print()
```

3. 树的转化

括号嵌套树 \Rightarrow 正常的多叉树

```
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.children = []
def build_Tree(string):
    node = None
    stack = [] # 及时处理
```

```
    for chr in string:
        if chr.isalpha(): # 这个是一个判断函数，多见于buffer
            node = TreeNode(chr)
            if stack:
                stack[-1].children.append(node)
        elif chr == "(":
            stack.append(node)
            node = None # 及时更新
        elif chr == ")":
            node = stack.pop() # 最后返回树根
        else:
            continue
    return node
# stack在这里的运用非常符合栈的定义和特征
def preorder(root):
    output = [root.val]
    for i in root.children: # 这里的输出不一样，因为孩子不止一个
        output.extend(preorder(i))
    return "".join(output)
```

括号嵌套树 \Leftarrow 正常的多叉树

```
def convert_to_bracket_tree(node):
    # 两个终止条件
    if not node:
        return ""
    if not node.children:
        return node.val
    result = node.val + "("
    for i, child in enumerate(node.children):
        result += convert_to_bracket_tree(child)
        if i != len(node.children) - 1:
            result += "," # 核心是“，”的加入，这里选择在一层结束前加入
    result += ")"
    return result
```

文件转化树

```
class Dir:
    def __init__(self, file_name):
        self.name = file_name
        self.dirs = []
        self.files = []
    def show(self, dir_name, layers = 0): # 这里把layer作为遍历的“线”
        layer = layers
        result = ["|     " * layer + dir_name.name]
        dir_name.files.sort()
        for dir in dir_name.dirs:
            result.extend(self.show(dir, layer + 1))
```

```
        for file in dir_name.files:
            result.extend(["|     " * layer + file]) # extend(str)会把字符串拆开
        return result
n = 0
while True:
    n += 1
    stack = [Dir("ROOT")] # 这的输入比较难，其实也是采用的是栈的思路—及时处理，及时退出
    while (s := input()) != "*":
        if s == "#":
            exit()
        if s[0] == "f":
            stack[-1].files.append(s)
        elif s[0] == "d":
            stack.append(Dir(s))
            stack[-2].dirs.append(stack[-1])
        else:
            stack.pop()
    print(f"DATA SET {n}:")
    print(*stack[0].show(stack[0]), sep="\n") # result是个列表，存储字符串
    print() # 分割线
```

4.树的分类

1. 解析树 ParseTree & 抽象语法树，AST

构建解析树的第一步是将表达式字符串拆分成标记列表。需要考虑4种标记：==左括号、右括号、运算符和操作数==。

```
class BinaryTree:
    def __init__(self, rootObj):
        self.key = rootObj
        self.leftChild = None
        self.rightChild = None
    def insertLeft(self, newNode):
        if self.leftChild == None:
            self.leftChild = BinaryTree(newNode)
        else:  # 已经存在左子节点。此时，插入一个节点，并将已有的左子节点降一层。
            t = BinaryTree(newNode)
            t.leftChild = self.leftChild
            self.leftChild = t
    def insertRight(self, newNode):
        if self.rightChild == None:
            self.rightChild = BinaryTree(newNode)
        else:
            t = BinaryTree(newNode)
            t.rightChild = self.rightChild
            self.rightChild = t
    def getRightChild(self):
        return self.rightChild
    def getLeftChild(self):
```

```
        return self.leftChild
    def setRootVal(self, obj):
        self.key = obj
    def getRootVal(self):
        return self.key
    def traversal(self, method="preorder"):
        if method == "preorder":
            print(self.key, end=" ")
        if self.leftChild != None:
            self.leftChild.traversal(method)
        if method == "inorder":
            print(self.key, end=" ")
        if self.rightChild != None:
            self.rightChild.traversal(method)
        if method == "postorder":
            print(self.key, end=" ")
def buildParseTree(fpexp):
    fplist = fpexp.split()
    pStack = [] # 其实就是stack
    eTree = BinaryTree('')
    pStack.push(eTree)
    currentTree = eTree
    for i in fplist:
        if i == '(':
        # (1) 如果当前标记是(，就为当前节点添加一个左子节点，并下沉至该子节点；
            currentTree.insertLeft('') # 未命名的树结点
            pStack.append(currentTree) # 压回parent结点
            currentTree = currentTree.getLeftChild() # 指针传到左结点
        elif i not in '+-*/)':
        # (3) 如果当前标记是数字，就将当前节点的值设为这个数并返回至父节点；
            currentTree.setRootVal(int(i)) # 命名叶子结点的值
            parent = pStack.pop() # 回到运算符
            currentTree = parent # 指针回溯
        elif i in '+-*/':
        # (2) 如果当前标记在列表`['+', '-', '/', '*']`中，就将当前节点的值设为当前 标记对应的运算符；为当前节点添加一个右子节点，并下沉至该子节点；
            currentTree.setRootVal(i) # 命名结点的运算符
            currentTree.insertRight('') # 开右子结点的空间
            pStack.append(currentTree) # 压回parent
            currentTree = currentTree.getRightChild() # 指针传到右结点
        elif i == ')':
        # (4) 如果当前标记是)，就跳到当前节点的父节点。
            currentTree = pStack.pop() # 该结点处理结束，弹出
        else:
            raise ValueError("Unknown Operator: " + i)
    return eTree
```

2. Huffman 算法

要构建一个最优的哈夫曼编码树，首先需要对给定的字符及其权值进行排序。然后，通过重复合并权值最小的两个节点（或子树），直到所有节点都合并为一棵树为止。下面是用 Python 实现的代码：

```
import heapq
class Node:
    def __init__(self, char, freq):
        self.char = char
        self.freq = freq
        self.left = None
        self.right = None
    def __lt__(self, other):
        return self.freq < other.freq
def huffman_encoding(char_freq):
    heap = [Node(char, freq) for char, freq in char_freq.items()]
    heapq.heapify(heap)
    while len(heap) > 1:
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)
        merged = Node(None, left.freq + right.freq) # note: 合并之后 char 字典是空
        merged.left = left
        merged.right = right
        heapq.heappush(heap, merged)
    return heap[0]
# 同样的 以depth作为递归深度的线
def external_path_length(node, depth=0):
    if node is None:
        return 0
    if node.left is None and node.right is None:
        return depth * node.freq
    return (external_path_length(node.left, depth + 1) +
            external_path_length(node.right, depth + 1))
def main():
    char_freq = {'a': 3, 'b': 4, 'c': 5, 'd': 6, 'e': 8, 'f': 9, 'g': 11, 'h': 12}
    huffman_tree = huffman_encoding(char_freq)
    external_length = external_path_length(huffman_tree)
    print("The weighted external path length of the Huffman tree is:",
external_length)
if __name__ == "__main__":
    main()
```

22161: 哈夫曼编码树 根据字符使用频率(权值)生成一棵唯一的哈夫曼编码树。生成树时需要遵循以下规则以确保唯一性： 选取最小的两个节点合并时，节点比大小的规则是:

1. 权值小的节点算小。权值相同的两个节点，字符集里最小字符小的，算小。
2. 合并两个节点时，小的节点必须作为左子节点
3. 连接左子节点的边代表0,连接右子节点的边代表1 然后对输入的串进行编码或解码 **输入** 第一行是整数n，表示字符集有n个字符。 接下来n行，每行是一个字符及其使用频率（权重）。字符都是英文字母。 再接下来是若干行，有的是字母串，有的是01编码串。 **输出** 对输入中的字母串，输出该字符串的编码 对输入中的01串,将其解码，输出原始字符串 样例输入

```
3
g 4
d 8
```

```
c 10
dc
110
```

样例输出

```
110
dc
```

建树：主要利用最小堆，每次取出weight最小的两个节点，weight相加后创建节点，连接左右孩子，再入堆，直至堆中只剩一个节点. 编码：跟踪每一步走的是左还是右，用0和1表示，直至遇到有char值的节点，说明到了叶子节点，将01字串添加进字典. 解码：根据01字串决定走左还是右，直至遇到有char值的节点，将char值取出.

```
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]
#以下把char 和密码对应上了
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
# 找到第一个字母为止
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
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 = []
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（Binary Search Tree, BST）

性质：小于父节点的键都在左子树中，大于父节点的键则都在右子树中。我们称这个性质为二叉搜索性。 给出一棵二叉搜索树的前序遍历，求它的后序遍历 **输入** 第一行一个正整数n（n<=2000）表示这棵二叉搜索树的结

点个数 第二行n个正整数，表示这棵二叉搜索树的前序遍历 保证第二行的n个正整数中，1~n的每个值刚好出现一次 **输出** 一行n个正整数，表示这棵二叉搜索树的后序遍历

```
class TreeNode:
    def __init__(self, value):
        self.val = value
        self.left = None
        self.right = None
def insert(root, value):
    if root is None:
        return TreeNode(value)
    elif value > root.val:
        root.right = insert(root.right, value)  # 这里是一个递归计算
    else:
        root.left = insert(root.left, value)
    return root
def postOrder(root):
    if root is None:
        return []
    return postOrder(root.left) + postOrder(root.right) + [root.val]
def inorder_traversal(root, result):
    if root:
        inorder_traversal(root.left, result)
        result.append(root.val)
        inorder_traversal(root.right, result)
N = int(input())
arr = list(map(int, input().split()))
root = None
for value in arr:
    root = insert(root, value)
print(*postOrder(root))
```

并查集

```
class DidjSet():
    def __init__(self,n):
        self.parent=list(range(n+1))
    def find(self,i):
        if self.parent[i]!=i:
            self.parent[i]=self.find(self.parent[i])
        return self.parent[i]
    def union(self,i,j):
        pari=self.find(i)
        parj=self.find(j)
        if pari!=parj:
            self.parent[pari]=parj
class disj_set:
    def __init__(self,n):#n表示并查集的初始大小
        self.rank = [1 for i in range(n)]
        #用于记录每个元素所在树的深度
```

```
        self.parent = [i for i in range(n)]
        #用于记录每个元素的父节点，初始时父节点都是自己
    def find(self,x):#用于查找X所在的集合，即数根节点（根节点条件：parent=自己本身）
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])#路径压缩：把父节点不停往根节
点找
        return self.parent[x]#返回最终的根节点
    def union(self,x,y):#用于合并x，y所在的两个集合
        x_root = self.find(x)#分别找到两者的根节点
        y_root = self.find(y)
        if x_root == y_root:
            return
    #哪颗树更深就把哪颗树设为主树，把另一棵树连到根节点
        if self.rank[x_root] > self.rank[y_root]:
            self.parent[y_root] = x_root
        elif self.rank[y_root] < self.rank[x_root]:
            self.parent[x_root] = y_root
        else:
            self.parent[y_root] = x_root
            self.rank[x_root] += 1
```

01182: 食物链 动物王国中有三类动物A,B,C，这三类动物的食物链构成了有趣的环形。A吃B， B吃C，C吃A。 现有N个动物，以1 - N编号。每个动物都是A,B,C中的一种，但是我们并不知道它到底是哪一种。 有人用两种说法对这N个动物所构成的食物链关系进行描述： 第一种说法是"1 X Y"，表示X和Y是同类。 第二种说法是"2 X Y"，表示X吃Y。 此人对N个动物，用上述两种说法，一句接一句地说出K句话，这K句话有的是真的，有的是假的。当一句话满足下列三条之一时，这句话就是假话，否则就是真话。 1） 当前的话与前面的某些真的话冲突，就是假话； 2） 当前的话中X或Y比N大，就是假话； 3） 当前的话表示X吃X，就是假话。 你的任务是根据给定的N（1 <= N <= 50,000）和K句话（0 <= K <= 100,000），输出假话的总数。 **输入** 第一行是两个整数N和K，以一个空格分隔。 以下K行每行是三个正整数 D，X，Y，两数之间用一个空格隔开，其中D表示说法的种类。若D=1，则表示X和Y是同类。若D=2，则表示X吃Y。 **输出** 只有一个整数，表示假话的数目。 样例输入

```
100 7
1 101 1
2 1 2
2 2 3
2 3 3
1 1 3
2 3 1
1 5 5
```

样例输出

```
3
```

我们设[0,n)区间表示同类，[n,2*n)区间表示x吃的动物，[2*n,3*n)表示吃x的动物。如果是关系1：将y和x合并，将y吃的与x吃的合并。将吃y的和吃x的合并。如果是关系2：将y和x吃的合并。将吃y的与x合并。将y吃的与吃x的合并。

```
# p = [0]*150001
def find(x):    # 并查集查询
    if p[x] == x:
        return x
    else:
        p[x] = find(p[x])   # 父节点设为根节点。目的是路径压缩。
        return p[x]
n,k = map(int, input().split())
p = [0]*(3*n + 1)
for i in range(3*n+1):  #并查集初始化
    p[i] = i
ans = 0
for _ in range(k):
    a,x,y = map(int, input().split())
    if x>n or y>n:
        ans += 1; continue
    if a==1:
        if find(x+n)==find(y) or find(y+n)==find(x):
            ans += 1; continue
        # 合并
        p[find(x)] = find(y)
        p[find(x+n)] = find(y+n)
        p[find(x+2*n)] = find(y+2*n)
    else:
        if find(x)==find(y) or find(y+n)==find(x):
            ans += 1; continue
        p[find(x+n)] = find(y)
        p[find(y+2*n)] = find(x)
        p[find(x+2*n)] = find(y+n)
print(ans)
```

堆

堆是一种特殊的树形数据结构，其中每个节点的值都小于或等于（最小堆）或大于或等于（最大堆）其子节点的值。堆分为最小堆和最大堆两种类型，其中：

- 最小堆： 父节点的值小于或等于其子节点的值。
- 最大堆： 父节点的值大于或等于其子节点的值。 堆常用于实现优先队列和堆排序等算法。

```
import heapq
heapq.heapify(x) heapq.heapify_max(list)
###将列表转换为最小堆/最大堆。
heapq.heappushpop(heap, item)
##将 item 放入堆中，然后弹出并返回 heap 的最小元素。该组合操作比先调用 heappush() 再调用 heappop() 运行起来更有效率
heapq.heapreplace(heap, item)
##弹出并返回最小的元素，并且添加一个新元素item
heapq.heappop(heap,item)
heapq.heappush(heap,item)
```

BFS和DFS

升空的焰火，从侧面看

```
import copy
def bfs(treenode,root):
    rightview=[]
    queue=[root]
    while queue:
        rightview.append(queue[-1])
        queue1=[]
        for i in queue:
            for j in treenode[i]:
                if j != -1:
                    queue1.append(j)
        queue=copy.deepcopy(queue1)
    return rightview
n=int(input())
treenode={}
for i in range(n):
    leftnode,rightnode=map(int,input().split())
    treenode[i+1]=[leftnode,rightnode]
print(' '.join(str(k) for k in bfs(treenode,1)))
```

马走日

```
operation=[[-2,-1],[-2,1],[2,-1],[2,1],[-1,-2],[-1,2],[1,-2],[1,2]]
ans=0
def dfs(dep,x,y):
    if dep==n*m:
        global ans
        ans+=1
        return
    for k in range(len(operation)):
        a =x+ operation[k][0]
        b =y+ operation[k][1]
        if 0<=a<n and 0<=b<m and chess[a][b]==False:
            chess[a][b] = True
            dfs(dep+1,x+operation[k][0],y+operation[k][1])
            chess[a][b] = False
for _ in range(int(input())):
    n, m, x, y = map(int, input().split())
    chess = [[False] * m for _ in range(n)]
    ans=0
    chess[x][y] = True
    dfs(1, x, y)
    print(ans)
```

图

1. **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==等。这些表示方法影响着对==图进行操作和算法实现==的效率。
2. **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）和广度优先搜索（BFS）。
3. **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。这些算法在网络路由、路径规划等领域有广泛的应用。 **Dijkstra算法**：Dijkstra算法用于解决单源最短路径问题，即从给定源节点到图中所有其他节点的最短路径。算法的基本思想是通过不断扩展离源节点最近的节点来逐步确定最短路径。具体步骤如下：

- 初始化一个距离数组，用于记录源节点到所有其他节点的最短距离。初始时，==源节点的距离为0，其他节点的距离为无穷大==。
- 选择一个未访问的节点中距离最小的节点作为当前节点。
- 更新当前节点的邻居节点的距离，如果通过当前节点到达邻居节点的路径比已知最短路径更短，则==更新最短路径==。
- 标记当前节点为==已访问==。
- 重复上述步骤，直到所有节点都被访问或者所有节点的最短路径都被确定。 Dijkstra算法的时间复杂度为O(V^2)，其中V是图中的节点数。当使用优先队列（如最小堆）来选择距离最小的节点时，可以将时间复杂度优化到O((V+E)logV)，其中E是图中的边数。

```
# 03424: Candies
# http://cs101.openjudge.cn/practice/03424/
import heapq
def dijkstra(N, G, start):
    INF = float('inf')
    dist = [INF] * (N + 1)  # 存储源点到各个节点的最短距离
    dist[start] = 0  # 源点到自身的距离为0
    pq = [(0, start)]  # 使用优先队列，存储节点的最短距离
    while pq:
        d, node = heapq.heappop(pq)  # 弹出当前最短距离的节点
        if d > dist[node]:  # 如果该节点已经被更新过了，则跳过
            continue
        for neighbor, weight in G[node]:  # 遍历当前节点的所有邻居节点
            new_dist = dist[node] + weight  # 计算经当前节点到达邻居节点的距离
            if new_dist < dist[neighbor]:  # 如果新距离小于已知最短距离，则更新最短距
离
                dist[neighbor] = new_dist
                heapq.heappush(pq, (new_dist, neighbor))  # 将邻居节点加入优先队列
    return dist
N, M = map(int, input().split())
G = [[] for _ in range(N + 1)]  # 图的邻接表表示
for _ in range(M):
    s, e, w = map(int, input().split())
    G[s].append((e, w))
start_node = 1  # 源点
shortest_distances = dijkstra(N, G, start_node)  # 计算源点到各个节点的最短距离
print(shortest_distances[-1])  # 输出结果
```

4. **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==。最小生成树在网络设计、电力传输等领域有着重要的应用。
5. **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。

```
#舰队，海域出击。思路：从一个节点开始，然后访问它的每一个邻居。如果在访问过程中，遇到了一个已经在当前路径中的节点，那么就存在一个环。可以使用一个颜色数组来跟踪每个节点的状态：未访问（0），正在访问（1），已访问（2）。
def has_cycle(n,edges):
    graph = [[] for _ in range(n)]
    for u, v in edges:
        graph[u].append(v)
    color=[0]*n
    def dfs(node):
        if color[node] == 1:
            return True
        if color[node] == 2:
            return False
        color[node] = 1
        for neighbor in graph[node]:
            if dfs(neighbor):
                return True
        color[node] = 2
        return False
    for i in range(n):
        if dfs(i):
            return 'Yes'
    return 'No'
T=int(input())
for _ in range(T):
    N,M=map(int,input().split())
    edges=[]
    for i in range(M):
        x,y=map(int,input().split())
        edges.append((x-1,y-1))
    print(has_cycle(N,edges))
```

7. **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。这对于网络分析、社交网络分析等具有重要意义。