Cheatsheet

by 信息科学技术学院 吴一凡 2024.6

一.排序

(1) 冒泡排序

(2) 插入排序

(3) 选择排序

(4) 快速排序

```
def quickSort(numList):
    n = len(numList)
    if n == 0 or n == 1:
        return numList
    pivot = numList[0]
    left = []
    right = []
    for i in range(1,n):
        if numList[i] < pivot:
            left.append(numList[i])
        else:
            right.append(numList[i])
    return quickSort(left) + [pivot] + quickSort(right)
```

(5) 希尔排序（缩小增量）

```
def  shellSort(numList):
    n = len(numList)
    if n == 0 or n == 1:
        return numList
    gap = n // 2
    while gap >= 1:
        for i in range(gap,n):
把前gap个空出来，以便进行各组之间的插入排序
            tempindex = i
            while tempindex >= gap and numList[tempindex - gap] > 
numList[tempindex]:
                numList[i - gap], numList[tempindex] = 
numList[tempindex],numList[tempindex - gap]
                tempindex -= gap
                # 先把一个子序列中的元素排好序，子序列中的元素下标之间的间隔为gap
        gap = gap // 2
    return numList
```

（6）堆排序

首先将待排序的数组构造出一个大顶堆，取出这个大顶堆的堆顶节点，与堆的最下最右的元素进行交换，然后把剩下的元素再构造出一个大根堆。重复第二步，直到这个大根堆的长度为1，此时完成排序

```python
global length
def buildMaxHeap(numList):
    global length
    length = len(numList)
    for i in range(length//2,-1,-1):
        heapify(numList,i)
def heapify(numList,i):
    global left,right,largest,length
    leftChildren = 2 * i + 1
    rightChildren = 2 * i + 2
    largest = i
    if leftChildren < length and numList[leftChildren] > numList[largest]:
        largest = leftChildren
    if rightChildren < length and numList[rightChildren] > numList[largest]:
        largest = rightChildren
    if largest != i:
        swap(numList,i,largest)
        heapify(numList,largest)
def swap(numList,i,j):
    numList[i],numList[j] = numList[j],numList[i]
def heapSort(numList):
    global length
    buildMaxHeap(numList)
    for i in range(len(numList)-1,0,-1):
        swap(numList,0,i)
        length -= 1
        heapify(numList,0)
    return numList
numlist = [4,5,4,1,8,7,2,6,3]
print(heapSort(numlist))
```

（7）归并排序

```python
def mergeSort(numList):
    if len(numList) == 0 or len(numList) == 1:
        return numList
    mid = len(numList) // 2
    left = numList[:mid]
    right = numList[mid:]
    sortedLeft = mergeSort(left)
    sortedRight = mergeSort(right)
    return merge(sortedLeft,sortedRight)
def merge(left,right):
    result = []
    while len(left) > 0 and len(right) > 0:
        if left[0] <= right[0]:
            result.append(left.pop(0))
        else:
            result.append(right.pop(0))
    result += right
    result += left
    return result
```

(8) 拓扑排序

```
import heapq
def solve(mat):
    n=len(mat)
    h=[]
    ru=[len(mat[i]) for i in range(n)]
    ans=[]
    for i in range(n):
        if ru[i]==0:
            heapq.heappush(h,i)
    while h:
        x=heapq.heappop(h)
        ans.append(x)
        for y in mat[x]:
            ru[y]-=1
            if ru[y]==0:
                heapq.heappush(h,y)
    if len(ans)<n:
        return -1
    else:
        return ans
```

二.栈

栈是后进先出的，队列是先进先出的

(1) 单调栈

例：给定一个列表，输出每个元素之前小于它的最后一个元素的下标

```
def solve(lis):
    n=len(lis)
    stack=[]
    ans=[]
    for i in range(n):
        x=lis[i]
        while stack:
            (y,j)=stack[-1]
            if y>=x:
                stack.pop()
                continue
            break
        if not stack:
            stack.append((x,i))
            ans.append(-1)
        else:
            ans.append(stack[-1][1])
            stack.append((x,i))
    return ans
```

（2）中序表达式转后序表达式

```
#shunting yard调度场算法
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]]:#判断优先级
                    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))
```

三.类

（1）类的改造

```
class person():
    #self 表示类的实例
    def __init__(self,a):
    self.name=a
    self.age=10
print(person(a).name)
```

(2) 用类实现双端队列

```
class deque:
    def __init__(self):
        self.queue=[]
    def push(self,a):#进队
        self.queue.append(a)
    def post_out(self):
        self.queue.pop()
    def pre_out(self):
        self.queue.pop(0)
    def empty(self):
        if self.queue==[]:
            return False
        else:
            return True
t=int(input())
for i in range(t):
    p=deque()
    n=int(input())
    for j in range(n):
            t,x=map(int,input().split())
            if t==1:
                p.push(x)
            elif t==2:
                if x==0:
                    p.pre_out()
                elif x==1:
                    p.post_out()
    if p.empty():
        ans=[]
        for z in p.queue:
            ans.append(str(z))
        print(' '.join(ans))
    else:
            print('NULL')
```

四.树

树由节点及连接节点的边构成，有如下性质：

1.有一个根节点；

2.除根节点外，其他每个节点都与其唯一的父节点相连；

3.从根节点到其他每个节点都有且仅有一条路径；

二叉树：每个节点最多有两个子节点

按形态分类:

（1）完全二叉树——第n-1层全满，最后一层按顺序排列（靠左）

（2）满二叉树——二叉树的最下面一层元素全部满就是满二叉树

（3）avl树——左右子树高度差不超过1

（4）二叉查找树(二叉排序\搜索树)——左<中<右

prim最小生成树：

从任意一个顶点开始，逐步扩展生成树，每次选择与生成树相连且权值最小的边加入生成树

```
import heapq
def solve(mat):
    h=[(0,0)]
    n=len(mat)
    vis=[1 for i in range(n)]
    ans=0
    while h:
        (d,x)=heapq.heappop(h)
        if vis[x]:
            vis[x]=0
            ans+=d
            for (y,t) in mat[x]:
                if vis[y]:
                    heapq.heappush(h,(t,y))
    return ans
```

kruskal最小生成树：

将所有边按权值排序，依次选择权值最小的边，若该边不形成环则加入生成树，直到树包含所有顶点。

```
import heapq
p=[i for i in range(n)]
def find(x):
    if p[x]==x:
        return x
    p[x]=find(p[x])
    return p[x]
def union(x,y):
    u,v=find(x),find(y)
    p[u]=v
def solve(mat):
    h=[]
    ans=0
    for i in range(n):
        for (j,d) in mat[i]:
            heapq.heappush(h,(d,i,j))
    while h:
        (d,i,j)=heapq.heappop(h)
        if find(i)!=find(j):
            union(i,j)
            ans+=d
    return ans
```

哈夫曼编码树

一种用于无损数据压缩的最优二叉树。它在信息编码过程中，以最小的平均编码长度实现对字符的编码

```
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
    #首先检查两个对象的 weight 属性是否相等。如果相等，就再比较它们的 char 属性，返回
self.char < other.char 的结果。如果不相等，就直接返回 self.weight < other.weight 的结果
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)#把两个最小节点构建一个融合的新节点
        merged.left = left
        merged.right = right
        heapq.heappush(heap, merged)
    return heap[0]
def encode_huffman_tree(root):
    #编码函数：使用递归方法遍历哈夫曼树的所有节点，并为每个叶子节点（即带有字符的节点）生成对应的
编码
    codes = {}
    def traverse(node, code):
        if node.char:
            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):
    #解码函数：从根节点开始，根据编码的每一位0或1向左或向右遍历树，直到找到叶子节点，即带有字符
的节点。将找到的字符添加到解码后的字符串中，继续开始下一轮遍历，直至解码完毕。
    decoded = ''
    node = root
    for bit in encoded_string:
        if bit == '0':
            node = node.left
        else:
            node = node.right
        if node.char:
            decoded += node.char
            node = root
    return decoded
n = int(input())
characters = {}
for _ in range(n):
    char, weight = input().split()
    characters[char] = int(weight)
huffman_tree = build_huffman_tree(characters)
codes = encode_huffman_tree(huffman_tree)
strings = []
while True:
    try:
        line = input()
        if line:
            strings.append(line)
        else:
            break
    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)
```

五.图

（1）判断无向图是否连通（有回路）

```
class Node:
def __init__(self, v):
        self.value = v
        self.joint = set()
"""判断是否连通"""
def connected(x, visited, num):
    visited.add(x)
    al = 1
    q = [x]
    while al != num and q:
        x = q.pop(0)
        for y in x.joint:
            if y not in visited:
                visited.add(y)
                al += 1
                q.append(y)
    return al == num
"""判断是否有环"""
def loop(x, visited, parent):
    visited.add(x)
    if x.joint:
        for a in x.joint:
            if a in visited and a != parent:
                return True
            elif a != parent and loop(a, visited, x):
                return True
    return False
n, m = map(int, input().split())
vertex = [Node(i) for i in range(n)]
```

```
for i in range(m):
    a, b = map(int, input().split())
    vertex[a].joint.add(vertex[b])
    vertex[b].joint.add(vertex[a])
if connected(vertex[0], set(), n):
    print('connected:yes')
else:
    print('connected:no')
x=0
for i in range(n):
    if loop(vertex[i],set(),None):
        print('loop:yes')
        x=1
        break
if x==0:
    print('loop:no')
```

（2）判断有向图是否有环：拓扑排序

原理：每次选入度为0的点，将该点放入output，并删掉该点的出边，同时更新其他点的入度。

如果最后output的长度为n则说明无环

```
class Node:
    def __init__(self, v):
        self.val = v
        self.to = []
from collections import deque
t = int(input())
for _ in range(t):
    n, m = map(int, input().split())
    node = [Node(i) for i in range(1, n + 1)]
    into = [0 for _ in range(n)]
    for _ in range(m):
        x, y = map(int, input().split())
        node[x - 1].to.append(node[y - 1])
        into[y - 1] += 1
    queue = deque([node[i] for i in range(n) if into[i] == 0])
    output = []
    while queue:
        a = queue.popleft()
        output.append(a)
        for x in a.to:
            num = x.val
            into[num - 1] -= 1
            if into[num - 1] == 0:
                queue.append(x)
    if len(output) == n:
        print('No')
    else:#否则说明有环
        print('Yes')
```

(3）BFS广度优先搜索

```
"""例题:最大连通域面积"""
t = int(input())
result = []
for _ in range(t):
    re = []
    ans = 0
    N, M = map(int, input().split())
    ma = []
    for _ in range(N):
        list1 = list(input())
        ma.append(list1)
    for i in range(N):
        for j in range(M):
            if ma[i][j] == 'W':
                ans += 1
                ma[i][j] = '.'
                queue = [(i, j)]
                while queue:
                    x, y = queue.pop()
                    for dx in [-1, 0, 1]:
                        for dy in [-1, 0, 1]:
                            nx, ny = x + dx, y + dy
                            if 0 <= nx < N and 0 <= ny < M and ma[nx][ny] == 'W':
                                ma[nx][ny] = '.'
                                queue.append((nx, ny))
                                ans += 1
            if ans > 0:
                re.append(ans)
                ans = 0
    re.append(ans)
    result.append(max(re))
for z in result:
    print(z)
```