

# 基础工具箱

## 1.1 字符串

```
s1.find(s2,start)#从start开始找，在s1中找s2序列，如果没有则会输出-1
for i,j in enumerate(iteration,start):#此循环，j在iteration中遍历(列表，字典)，i是参数，需要自行设置，默认为0，每次加1
s.upper();s.lower();s.capitalize()#大写，小写，首字母大写
ord() # 字符转ASCII
chr() # ASCII转字符
```

## 1.2 列表

```
s.sort(cmp=None, key=None, reverse=False)
#cmp是可选参数，指定后按照该参数的方法进行排序
#key是指定用来比较的参数，如果列表中的元素是自定义树的节点，则可以用key=node.value按照节点大小进行排序
#reverse=True降序 reverse=False升序（默认）
```

## 1.3 字典

```
dic={}#空字典
dic.keys();dic.values()#要想获得对应的列表需要前置list，但是可以判断 n in dic.values()
for n in dic#遍历字典的键
for i,j in dic.items()#获得键值对的元组
```

## 1.4 输出格式

```
print(*s)#s是列表，将其中元素全部输出，以' '分隔
print('%.1f' % x)#x输出一位小数
print(f'Case {c}')#c替换为c的值
```

树的输出列在下面的代码清单中。

```
def preorder(node):#前序遍历函数，node是根节点,node具有属性node.children=[]
    output=[node.value]
    for child in node.children:
        output.extend(preorder(child))#递归地遍历子树，已经保证从左开始
    return ''.join(output)
def postorder(node):#后序遍历函数
    #node是根节点，此处给一个二叉树的例子，定义的类初始node.left=None;node.right=None
    ans=[]
    if node:
        if node.left:
            ans.extend(postorder(node.left))
        if node.right:
            ans.extend(postorder(node.right))
        ans.append(node.value)
    return ''.join(ans)
def level_order_traversal(root):#层次遍历,traversal就有“遍历”的意思，以二叉树为例
    queue=[root]
```

```

traversal=[]
while queue:
    node=queue.pop(0)
    traversal.append(node.value)
    if node.left:
        queue.append(node.left)
    if node.right:
        queue.append(node.right)
return traversal

```

# 巨人的肩膀

## 2.1 itertools

```

from itertools import*
a=groupby(s)#s是一个字符串或者列表
for i,j in a:
    print(i,len(list(j)))#输出元素，及连续的个数，例如输入aaabbbaa输出a 3\\b 2\\a 2
for i in permutations(s,r):#i是s中所有可能的长度为r的排列
    print(''.join(map(str,i)))#把每一种可能输出
#product
from itertools import product
for s in product(range(n),repeat=N)#s是长度为N，每个位置有0~n-1共n-1种可能的序列

```

## 2.2 collections

```

from collections import*
a=Counter(s)#输出s中元素及其对应的个数，是以字典的形式
num=a[word]#num就是列表s中word出现的次数，这个计算比s.count(word)要快
d=deque()#可用len(deque),和list,str一样可以用iterable.count(x)计数x出现次数复杂度O(n)
d.pop() '右弹出' d.popleft() '左弹出'
d.append() '右添加' d.appendleft() '左添加'
d.extend() '右扩展' d.extendleft() '左扩展'
d.rotate()#将双向队列向右旋转 n 步。n 为正，右侧的元素会移动到左侧；n 为负，相反

```

## 2.3 math

```

from math import*
comb(n,k)#从n中无顺序选择k项的方式

```

## 2.4 copy

```

from copy import deepcopy
matrix=deepcopy(matrix_backup)#高维数组需要使用深拷贝

```

## 2.5 heapq

```

from heapq import*
heapify(x)#将列表x转化为最小堆
heappush(s,item)#将item的值加入s中，保持仍为最小堆
heappop(s)#弹出并返回堆最小元素
heap[0]#返回最小值而不弹出
heappushpop(s,item)#将item放入堆中，然后弹出返回s最小元素
heapreplace(s,item)#先弹出，再返回

```

## 2.6 bisect

```

from bisect import*
a=bisect_left(s,x)#不插入，返回的是如果插入，插入后的位置
a=bisect_right(s,x)
insort_left(s,x)#没有值，但是插入
insort_right(s,x)

```

## 2.7 functools

```

from functools import cmp_to_key
def cmp(x,y):
    a=
    b=#自定义比较条件
    return 1 if a>b else -1 if a<b else 0
s.sort(key=cmp_to_key(cmp))#将cmp(x,y)函数作为比较判准

```

# 美丽的画作

## 3.1 中序表达式转后序表达式

```

def infix_to_postfix(expression):
    precedence={'+':1,'-':1,'*':2,'/':2}
    stack=[]#运算符栈
    postfix=[]#输出栈
    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]]:
                    #注意是<=号，这和后序表达式逻辑有关
                    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))

```

## 3.2 字典树

```

class Trie:
    def __init__(self):
        self.root={}
        self.end=-1
    def add(self,word):
        node=self.root
        for letter in word:
            if letter not in node:
                node[letter]={}#如果不在键中，意味着产生了新的分支，建立新的字典
            node=node[letter]#沿着子树接着走
        node[self.end]=True#代表此处为一个单词的结束位置
    def search(self,word):#查找word是否在字典树中
        node=self.root
        for letter in word:
            if letter not in node:
                return False
            node=node[letter]
        if self.end not in node:#这个地方不是单词的结尾
            return False
        return True

```

## 3.3 Huffman 编码树

```

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):#定义了heapq的排序方法，不然会runtime error
        if self.weight==other.weight:
            if self.char and other.char:#比较大小注意None
                return self.char < other.char
            else:
                return True
        return self.weight<other.weight
def bulid_huffman_tree(characters):#字典，字符：权值
    heap=[]
    for char,weight in characters.items():#items返回可遍历的键值元组
        heapq.heappush(heap,Node(weight,char))

```

```

while len(heap)>1:
    left=heapq.heappop(heap)
    right=heapq.heappop(heap)
    merged=Node(left.weight+right.weight)#合并后, char为None
    merged.left=left
    merged.right=right
    heapq.heappush(heap,merged)
return heap[0]
def encode_huffman_tree(root):#确定编码值
    codes={}#字典, 存储每个字符对应的编码
    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):#对特定字符串编码
    encoded=''
    for char in string:
        encoded+=codes[char]
    return encoded
def huffman_decoding(root,encoded_string):#对01串解码
    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())
weights=list(map(int,input().split()))
character={}
for i in range(n):
    character[str(i)]=weights[i]
root=bulid_huffman_tree(character)
codes=encode_huffman_tree(root)
ans=0
for i in range(n):
    ans+=weights[i]*len(codes[str(i)])
print(ans)

```

### 3.3 Prim 算法

贪心。构建两个集合，一个是目前可达的点集A，一个是目前不可到达的点集B。每次从B中挑选最短的可以连接到A的路径添加。可以用来求最小生成树，具体情形中各种条件下最优的连通解。

```

import heapq
def prim(graph, start):
    mst = []
    used = set([start]) # 已经使用过的点

```

```

edges = [
    (cost, start, to)
    for to, cost in graph[start].items()
] # (cost, frm, to) 的列表
heapq.heapify(edges) # 转换成最小堆

while edges: # 当还有边可以选择时
    cost, frm, to = heapq.heappop(edges) # 弹出最小边
    if to not in used: # 如果这个点还没被使用过
        used.add(to) # 标记为已使用
        mst.append((frm, to, cost)) # 加入到最小生成树中
        for to_next, cost2 in graph[to].items(): # 将与这个点相连的边加入到堆中
            if to_next not in used: # 如果这个点还没被使用过
                heapq.heappush(edges, (cost2, to, to_next)) # 加入到堆中

    return mst # 返回最小生成树
n = int(input())
graph = {chr(i+65): {} for i in range(n)}
for i in range(n-1):
    data = input().split()
    node = data[0]
    for j in range(2, len(data), 2):
        graph[node][data[j]] = int(data[j+1])
        graph[data[j]][node] = int(data[j+1])
mst = prim(graph, 'A') # 从A开始生成最小生成树
print(sum([cost for frm, to, cost in mst])) # 输出最小生成树的总权值

```

### 3.4 拓扑队列 (Kahn算法)

可以用来判断有向图是否有环，同样也适用于无向图（入队条件变为度<=1）

```

def find(nodes): # nodes为储存顶点的队列
    queue = deque()
    result = []
    for node in nodes:
        if node.indegree == 0:
            queue.append(node)
    while queue:
        u = queue.popleft() # 需要调用collections里的deque
        result.append(u)
        for v in u.reach:
            v.indegree -= 1
            if v.indegree == 0:
                queue.append(v)
    if len(result) == n:
        return result # 返回拓扑序列
    return False # 有环

```

### 3.5 二分查找

用来处理一些隔板差值问题，如果正向很难想通也可以试试。（下面是河中跳房子）

```

l, n, m = map(int, input().split())
stone = [0]
for i in range(n):
    stone.append(int(input()))

```

```

stone.append(1)
def check(L,m,stone):
    distance=0
    off=0
    for i in range(1,n+2):
        distance+=stone[i]-stone[i-1]
        if distance<L:
            off+=1
        else:
            distance=0
    if off<=m:
        return off
    return False
mi,ma=0,1
ans=-1
while mi!=ma:
    middle=(mi+ma)//2
    if check(middle,m,stone):
        mi=middle+1
        ans=middle#注意具体案例中，查找过程的设置。此处若check(middle)，则middle可能就是
        答案
    else:
        ma=middle
print(ans)

```

### 3.6 单调栈

给出项数为  $n$  的整数数列  $a_1, \dots, a_n$ ，定义函数  $f(i)$  代表数列中第  $i$  个元素之后第一个大于  $a_i$  的元素的下标。若不存在，则  $f(i) = 0$ 。试求出  $f(1), \dots, f(n)$

```

n = int(input())
a = list(map(int, input().split()))
stack = []
for i in range(n):
    while stack and a[stack[-1]] < a[i]:
        #stack里的储存的位置必定是单调减小的，所以若不大于最后一个，则也不大于其中任意一个
        a[stack.pop()] = i + 1
    stack.append(i)
while stack:
    a[stack[-1]] = 0
    stack.pop()
print(*a)

```

## 最后的一步

### 4.1 乱七八糟

- (1) 检查输入和输出都正确吗？单纯的字符串输入加上strip()
- (2) 设计数据验证每个条件，尤其是边界情况。

## 4.2 树

## 4.3 图

- (1) 检查输入的边是有向的还是无向的？无向的边是否添加了两次？
- (2) 超时可以把visited改成多维数组。
- (3) runtime error检查排序的过程中是否涉及到自己定义的类。