算法

动归

Kadane算法——求解最大连续子数组和

对于一个线性列表a，维护两个变量s1和s2。其中s1记录了选当前列时最大的和，后者记录了选到当前列的最大和。

```
a = [1, -2, 3, 5, -4, 10, -22, 4, 22, -4, 0, -14]
s1 = 0
s2 = a[1]
for i in a:
    s1 = max(s1 + i, i)
    s2 = max(s1, s2)
print(s2)
```

前缀和优化

在进行线性连续的大量累加运算的时候，可以提前计算从头开始的部分和，计算其他部分和只需要用两个前缀和相减。
包括反向的差分。

欧拉筛

```
def oula(a):
    zhishu=[]
    zhishu1=[True]*(a+1)
    for i in range(2,a+1):
        if zhishu1[i]:
            zhishu.append(i)
        for h in zhishu:
            if h*i<=a:
                zhishu1[h*i]=False
    zhishu=set(zhishu)
    return zhishu
```

字符匹配算法KMP

```
def kmp(s1,s2):
    n,m=len(s1),len(s2)
    x,y=0,0
    nt=nextarray(s2,m)
    while x<n and y<m:
        if s1[x]==s2[y]:
            x+=1
            y+=1
        elif y==0:
            x+=1
        else:
            y=nt[y]
    return x-y if y==m else -1
def nextarray(s,m):
    # 返回一个列表a，其中a[i]记录了s[:i + 1]的最长公共真前后子缀的长度-1
    if m==1:
        return [-1]
    nt=[0]*m
    nt[0],nt[1]=-1,0
    i,cn=2,0
    while i<m:
        if s[i-1]==s[cn]:
            cn+=1
            nt[i]=cn
            i+=1
        elif cn>0:
            cn=nt[cn]
        else:
            nt[i]=0
            i+=1
    return nt
```

优先队列

使用最小堆实现

```
# 这是个人实现
class BinaryHeap:
    def __init__(self):
        self.heap = [None]
        self.size = 0

    def put(self, num):
        self.heap.append(num)
        self.size += 1
        self.float(self.size)
        return None

    def float(self, ind):
        while ind > 1:
            if self.heap[ind] < self.heap[ind // 2]:
                self.heap[ind], self.heap[ind // 2]\
                     = self.heap[ind // 2], self.heap[ind]
            else:
                break
            ind //= 2
        return None

    def pop(self):
        num = self.heap[1]
        if self.size > 1:
            self.heap[1] = self.heap.pop()
            self.size -= 1
            self.sink(1)
        else:
            self.heap.pop()
            self.size -= 1
        return num

    def sink(self, ind):
        while ind * 2 <= self.size:
            needbreak = True
            if self.heap[ind] > self.heap[ind * 2]:
                self.heap[ind], self.heap[ind * 2] = \
                    self.heap[ind * 2], self.heap[ind]
                ind1 = ind * 2
                needbreak = False
            if ind * 2 + 1 <= self.size and self.heap[ind] \
                > self.heap[ind * 2 + 1]:
```

```
                if needbreak:
                    self.heap[ind], self.heap[ind * 2 + 1] \
                        = self.heap[ind * 2 + 1], self.heap[ind]
                else:
                    # 已经调换过
                    self.heap[ind], self.heap[ind * 2] = \
                        self.heap[ind * 2], self.heap[ind]
                    self.heap[ind], self.heap[ind * 2 + 1] \
                        = self.heap[ind * 2 + 1], self.heap[ind]
                ind1 = ind * 2 + 1
                needbreak = False
            if needbreak:
                break
            ind = ind1
        return None
```

而heapq库可以解决这个问题

```
import heapq
heapq.heappush(heap, item) # 将元素item添加到列表heap中，并保持堆的特性。

heapq.heappop(heap) # 从堆heap中弹出并返回最小的元素，同时保持堆的特性。

heapq.heapify(x) # 将列表x转换成一个堆，这是一个原地操作，时间复杂度为线性。

heapq.heappushpop(heap, item) # 将元素item放入堆中，然后弹出并返回堆中
# 的最小元素。这个组合操作比单独调用heappush()后再调用heappop()更高效。

heapq.heapreplace(heap, item) # 弹出并返回堆中最小的元素，同时将新元素
# item推入堆中。这个操作在使用固定大小的堆时非常适用。
```

图的拓扑排序

1. 利用图的DFS实现，将节点按结束时间从大向小排序就可以得到，先做什么后做什么的线性排序。
2. 每次选出入度为0的点，然后更新其相连节点的入度。
   如果某一步有不止一个入度为0的点，说明排序不唯一
   如果某一步还没有排完所有的点，但是没有入度为0的点，说明有圈。

各种排序

冒泡

稳定；时间$O(n^2)$；空间$O(1)$

选择

不稳定；时间$O(n^2)$；空间$O(1)$

是冒泡的优化，比较次数相同但是交换次数减少，多用了一个变量来缓存。

插入

每次通过二分等方法为新的数据寻找位置，得到排列完成的子列表。

稳定；

谢尔

将列表分为间隔等距的一些子列表，分别插入排序，再整体插入排序

不稳定；

归并

分成两个子列表，排序完成后再合并

稳定；

快速

找到中值，把比他小的放一边，比他大的放一边

不稳定；

考察中值点的选择和不同的实现：霍尔法、三指针法、挖坑法

```
def quicksort(arr, left, right):
    if left < right:
        mid = partition(arr, left, right)
        quicksort(arr, left, mid - 1)
        quicksort(arr, mid + 1, right)
def partition(arr, left, right):
    i = left
    j = right - 1
    pivot = arr[right]
    while i <= j:
        while i <= right and arr[i] < pivot:
            i += 1
        while j >= left and arr[j] >= pivot:
            j -= 1
        if i < j:
            arr[i], arr[j] = arr[j], arr[i]
    if arr[i] > pivot:
        arr[i], arr[right] = arr[right], arr[i]
    return i
```

分配

语法

常用的库

```
# 堆
import heapq
# 队列(default字典)
from collections import deque(defaultdict)
# 递归上限
from sys import setrecursionlimit
# 缓存
from functools import lru_cache
'''数学***math库***： 最常用的sqrt,对数log(x[,base])、 三角sin()、 反三角
asin()也都有； 还有e,pi等常数， inf表示无穷大； 返回小于等于x的最大整数
floor（）,大于等于ceil（） ,判断两个浮点数是否接近
isclose（a， b， *,rel_tol=1e-09, abs_tol=0.0）；一般的取幂pow（x， y）
,阶乘factorial（x） 如果不符合会ValueError,组合数comb（n， k）
`math.radians()`将度数转换为弧度，`math.degrees()`将弧度转换为度数。'''
import math
# 二分库
import bisect
bisect.bisect_right(a,6)
# 返回在a列表中若要插入6的index（有重复数字会插在右边）
bisect.insort(a,6)# 返回插入6后的列表a
# conuter
from collections import Counter
```

类的定义方法

init 构造函数，创建对象时自动调用
del 析构函数，对象删除前调用
str 控制 print(obj) 时的输出
repr 控制对象在解释器中的表现
len 支持 len(obj)
getitem 支持 obj[key]

setitem 支持 obj[key] = value
iter 使对象可迭代（如用于 for 循环）
next 支持迭代器的下一个元素
call 使对象可以像函数一样调用
enter / **exit** 用于上下文管理器（with 语句）
eq , **lt** , **gt** , **ne** , **le** , **ge** 比较运算
add , **sub** , **mul** , 等等 支持算术运算符重载
eq(self, other)	==	判断相等
ne(self, other)	!=	判断不相等
lt(self, other)	<	判断是否小于
le(self, other)	<=	判断是否小于等于
gt(self, other)	>	判断是否大于
ge(self, other)	>=	判断是否大于等于

格式化字符串

未知行输入

```
import sys
for line in sys.stdin:
    line = line.strip()
    # 去掉每行最后的换行符，制表符和回车符
```

提示

二分法

二分试根的左右范围应当扩大，精度要求应该缩小。不要为了省事把第三种直接得到解的情况省略。左右指针精度要求应该强一些。

RE常见原因

1. 指针爆了
2. 数组太长爆了

3. 变量名错了
4. 栈爆了
   使用代码

```
from sys import setrecurisonlimit
setrecursionlimit(10000) # python 默认 200
```

模版

多项式加法和乘法

```
class node:
    def __init__(self, xishu, cishu):
        self.xishu = xishu
        self.cishu = cishu


class poly:
    def __init__(self, dan = None):
        self.pdict = dict()
        # 按次数储存系数
        if dan != None:
            self.pdict[dan.cishu] = dan

    def padd(self, dan):
        '''向多项式中添加单项式'''
        k = self.pdict.get(dan.cishu, node(0, dan.cishu))
        k.xishu += dan.xishu
        self.pdict[dan.cishu] = k
        return None

def add(a:poly, b:poly):
    k = poly()
    for i in a.pdict:
        k.padd(a.pdict[i])
    for i in b.pdict:
        k.padd(b.pdict[i])
    return k

def mul(a:poly, b:poly):
    k = poly()
    for i in a.pdict:
        i = a.pdict[i]
        for j in b.pdict:
            j = b.pdict[j]
            k.padd(node(i.xishu * j.xishu, i.cishu + j.cishu))
    return k
```

```
def strforpoly(a:poly):
    ind = list(a.pdict.keys())
    ind.sort(reverse=True)
    ans = ''
    for i in ind:
        x = a.pdict[i].xishu
        y = a.pdict[i].cishu
        if x != 0:
            if x > 0 and ans != '':
                ans += '+'
            if x == 1 and y != 0:
                pass
            elif x == -1 and y != 0:
                ans += '-'
            else:
                ans += str(x)
            if y != 0:
                ans += 'x'
                if y != 1:
                    ans += '^'
                    ans += str(y)
    if ans == '':
        ans = '0'
    return ans
```

图与图的深搜

```
class VertBase:
    """
    有向带权图的顶点
    当自定义类的对象用作字典的键值时，缺省使用系统hash()函数，它基于id()函数实现，
    也就是说，一个对象的哈希值在它的生命周期期间是固定的。
    """
    def __init__(self, key):
        self.id = key
        self.connectedTo = {}
        self.inDegree = self.outDegree = 0

    def addNeighbor(self, nbr, weight=1):
        """
        添加从当前节点到nbr节点的有向边
        添加重复边会更新其权值，不影响节点的出/入度。
        """
        if nbr not in self.connectedTo:
            self.outDegree += 1
            nbr.inDegree += 1
        self.connectedTo[nbr] = weight

    def delNeighbor(self, nbr):
        """
        删除有向边
        """
        if nbr in self.connectedTo:
            self.outDegree -= 1
            nbr.inDegree -= 1
            del(self.connectedTo[nbr])

    def __str__(self):
        return f"{self.id}({self.inDegree}:{self.outDegree})-->{[x.id for x in self.connectedTo]

    __repr__ = __str__

    def getConnections(self):
        """
        取得所有指向的节点
        """
        return self.connectedTo.keys()
```

```
    def getId(self):
        return self.id

    def getWeight(self, nbr):
        """
        取得指向某节点边的权重
        """
        return self.connectedTo[nbr]

import os

class Graph:
    """
    有向带权图，不支持重复边
    """
    def __init__(self, Vert=VertBase, other=None):
        """
        Vert: 顶点类型，需要继承基础顶点Vertex
        other: 复制other中所有的顶点和边
        """
        assert issubclass(Vert, VertBase)
        self.Vertex = Vert
        self.vertList = {}    #其实是一个字典
        if other:
            for v in other:
                self.addVertex(v.getId())
                for u in v.getConnections():
                    self.addEdge(v.getId(), u.getId(),
                            v.getWeight(u))

    def T(self):
        """
        获取有向图的转置图
        """
        ng = Graph()
        for v in self:
            ng.addVertex(v.getId())
            for u in v.getConnections():
                ng.addEdge(u.getId(), v.getId())
        return ng

    def addVertex(self, key):
```

```
    if key not in self:
        self.vertList[key] = Vertex(key)
    return self.vertList[key]

def getVertex(self, key):
    try:
        return self.vertList[key]
    except KeyError:
        return None

def delVertex(self, key):
    """
    删除一个顶点，先删掉它所有的入边、出边，再删本身
    """
    if key in self:
        v = self.getVertex(key)
        for f in self:
            f.delNeighbor(v)
        for t in [t for t in v.getConnections()]:
            v.delNeighbor(t)
        del(self.vertList[key])

def __contains__(self, key):
    return key in self.vertList

def addEdge(self, f, t, weight=0):
    if f not in self:
        self.addVertex(f)
    if t not in self:
        self.addVertex(t)
    self.vertList[f].addNeighbor(self.vertList[t], weight)

def getVertices(self):
    """
    返回所有顶点的key
    """
    return self.vertList.keys()

def __iter__(self):
    """
    返回所有顶点本身
    """
    return iter(self.vertList.values())
```

```
    def __len__(self):
        return len(self.vertList)

    def __str__(self):
        """
        每个顶点各占一行。
        """
        s = ""
        for v in self:
            nbrs = " ".join([str(u.getId()) for u in v.getConnections()])
            s += f"{v}: {nbrs}{os.linesep}"
        return s

#对VertBase进行扩充，
#增加了distance, color, pred
#和discovery, finish属性，
#从而对BFS/DFS搜索算法进行支持。
import sys
class Vertex(VertBase):
    def __init__(self, key):
        super().__init__(key)
        self.color = 'white'
        self.distance = sys.maxsize
        self.pred = None
        """
        self.pred = []
        前驱可能有多个值，用一个列表来存放。
        这样可以通过BFS算法找出所有的最短路径。
        """
        self.discovery = None
        self.finish = None

    def setDiscovery(self, t):
        self.discovery = t

    def setFinish(self, t):
        self.finish = t

    def getDistance(self):
        return self.distance

    def setDistance(self, dist):
```

```
        self.distance = dist

    def setPred(self, pred):
        self.pred = pred

    def getPred(self):
        return self.pred

    def getColor(self):
        return self.color

    def setColor(self, color):
        self.color = color

# 图的深搜
class DFSGraph(Graph):
    def __init__ (self, other = None):
        super().init(Vertex, other)
        self.time = 0 # 不是物理时间，而是算法执行步数

    def dfs(self):
        for aVertex in self:
            aVertex.setColor('white') # 颜色初始化
            aVertex.setPred(None)
        # 从每一个顶点开始遍历
        for aVertex in self:
            if aVertex.getColor() == 'white':
                self.dfsvisit(aVertex) # 建立一个以aVertex为根的树。如果有多棵树，则形成森林

    def dfsvisit(self, startVertex):
        startVertex.setColor('gray')
        self.time += 1 # 记录算法的步数
        startVertex.setDiscovery(self.time)
        for nextVertex in startVertex.getConnections():
            if nextVertex.getColor()=='white':
                nextVertex.setPred(startVertex)
                self.dfsvisit(nextVertex) # 深度优先递归访问
        startVertex.setColor('black')
        self.time += 1
        startVertex.setFinish(self.time)
```

表达式树

目前会wa模考第六题，不知道哪里有问题，感觉正确，可以尝试使用

```
class node:
    def __init__(self, data):
        self.data = data
        self.leftson = None
        self.rightson = None

def buildtree(line):
    # 从中缀表达式建树
    num = 0
    i = 0
    numstack = []
    opstack = []
    pre_is_num = False
    while i < len(line):
        stack = []
        if line[i] != '(':
            if line[i] == '*':
                opstack.append('*')
                pre_is_num = False
            elif line[i] == '+':
                if not opstack:
                    opstack.append('+')
                    pre_is_num = False
                else:
                    x = opstack.pop()
                    if x == '+':
                        opstack.append('+')
                        opstack.append('+')
                        pre_is_num = False
                    else:
                        b = numstack.pop()
                        a = numstack.pop()
                        n = node('*')
                        n.leftson = a
                        n.rightson = b
                        numstack.append(n)
                        # 把应当优先计算的子表达式树压入numstack
                        opstack.append('+')
                        pre_is_num = False
            else:
                if pre_is_num:
                    x = numstack.pop()
                    x.data += line[i]
```

```
                    numstack.append(x)
                else:
                    numstack.append(node(line[i]))
                    pre_is_num = True
            i += 1
        else:
            num += 1
            j = i + 1
            while num:
                if line[j] == ')':
                    num -= 1
                elif line[j] == '(':
                    num += 1
                stack.append(line[j])
                j += 1
            stack.pop()
            numstack.append(buildtree(stack))
            # 得到了括号内的子表达式树，压入numstack
            i = j
            pre_is_num = False
    # numstack与opstack维护完毕，现在构建表达式树
    while opstack:
        op = opstack.pop()
        b = numstack.pop()
        a = numstack.pop()
        n = node(op)
        n.leftson = a
        n.rightson = b
        numstack.append(n)
    # 最后一个n就是整个树
    return numstack.pop()

def write(tree):
    if tree.data == '+':
        return write(tree.leftson) + '+' + write(tree.rightson)
    if tree.data == '*':
        ans = ''
        l = tree.leftson
        r = tree.rightson
        if l.data == '+':
            ans += '(' + write(l) + ')'
        else:
            ans += write(l)
```

```
    ans += '*'
    if r.data == '+':
        ans += '(' + write(r) + ')'
    else:
        ans += write(r)
    return ans
return tree.data
```