递归记忆化

```
from functools import lru_cache
@lru_cache(maxsize=None)
```

解除递归限制

```
import sys
sys.setrecursionlimit(10000)
```

树：

中序转后序（调度场算法）

```
n=int(input().strip())
for _ in range(n):
    s=input().strip()
    operator=[]
    output=[]
    ops={'+':0,'-':0,'*':1,'/':1}
    i=0
    l=len(s)
    while i<l:
        if '0'<=s[i]<='9':
            num=''
            while i<l and ('0'<=s[i]<='9' or s[i]=='.'):
                    num+=s[i]
                    i+=1
            i-=1
            output.append(num)
        elif s[i]=='(':
            operator.append('(')
        elif s[i]==')':
            while operator and operator[-1]!='(':
                output.append(operator.pop())
            operator.pop()
        else:
            while operator and operator[-1] in ops and
ops[operator[-1]]>=ops[s[i]]:
                output.append(operator.pop())
            operator.append(s[i])
        i+=1
    while operator:
        output.append(operator.pop())
    print(*output)
```

去除多余括号（先转后序，在建树，再打印）

```
class BinaryTree:
    def __init__(self, root, left=None, right=None):
        self.root = root
        self.leftChild = left
        self.rightChild = right

def postorder(string):  # 中缀改后缀 (Shunting Yard)
    opStack, postList = [], []
    inList = string.split()
    prec = {'(': 0, 'or': 1, 'and': 2, 'not': 3}
```

```
    for word in inList:
        if word == '(':
            opStack.append(word)
        elif word == ')':
            while opStack and opStack[-1] != '(':
                postList.append(opStack.pop())
            opStack.pop()
        elif word in ('True', 'False'):
            postList.append(word)
        else:  # operator
            while opStack and prec[word] <= prec[opStack[-1]]:
                postList.append(opStack.pop())
            opStack.append(word)
    while opStack:
        postList.append(opStack.pop())
    return postList

def buildParseTree(infix):
    postList = postorder(infix)
    stack = []
    for word in postList:
        if word == 'not':
            child = stack.pop()
            stack.append(BinaryTree('not', child))
        elif word in ('True', 'False'):
            stack.append(BinaryTree(word))
        else:
            right, left = stack.pop(), stack.pop()
            stack.append(BinaryTree(word, left, right))
    return stack[-1]

# 定义运算符优先级
priority = {'or': 1, 'and': 2, 'not': 3, 'True': 4, 'False': 4}

def printTree(tree):
    """返回 token 列表"""
    root = tree.root
    if root in ('True', 'False'):
        return [root]

    if root == 'not':
        child = tree.leftChild
        # 若子优先级更低则加括号
        child_tokens = printTree(child)
        if priority[child.root] < priority[root]:
            child_tokens = ['('] + child_tokens + [')']
        return ['not'] + child_tokens

    # 二元操作符 and/or
    left, right = tree.leftChild, tree.rightChild
    left_tokens = printTree(left)
    right_tokens = printTree(right)
    if priority[left.root] < priority[root]:
        left_tokens = ['('] + left_tokens + [')']
    if priority[right.root] < priority[root]:
        right_tokens = ['('] + right_tokens + [')']
    return left_tokens + [root] + right_tokens

def main():
    infix = input().strip()
```

```
    Tree = buildParseTree(infix)
    print(' '.join(printTree(Tree)))

if __name__ == "__main__":
    main()
```

哈夫曼编码树

```
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):
        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, min(left.char, right.char))#规则
        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.left is None and node.right is None:#是字母（叶）
                decoded += node.char
                node = root#从头再来
        return decoded

n = int(input())
characters = {}
for _ in range(n):
    char, weight = input().split()
    characters[char] = int(weight)
# 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()
        if line[0] in ('0','1'):
            print(huffman_decoding(huffman_tree, line))
        else:
            print(huffman_encoding(codes, line))

    except EOFError:
        break
```

给定前序和中序遍历，求后序遍历，可以不建立二叉树，直接输出

```
def f(preorder,inorder):
    if not preorder or not inorder:
        return ''
    root=preorder[0]
    root_index=inorder.index(root)
    left_inorder=inorder[:root_index]
    right_inorder=inorder[root_index+1:]
    left_preorder=preorder[1:len(left_inorder)+1]
    right_preorder=preorder[len(left_inorder)+1:]
    l=f(left_preorder,left_inorder)
    r=f(right_preorder,right_inorder)
    return l+r+root

try:
    while True:
        line = input().strip()
        preorder, inorder = line.split()
        print(f(preorder, inorder))
except EOFError:
    pass
```

扩展二叉树

```
class TreeNode:
    def __init__(self,val):
        self.val=val
```

```
        self.num=0
        self.left=None
        self.right=None
def build_tree(s):
    node=TreeNode(s[0])
    stack=[node]
    for a in s[1:]:
        if a=='.':
            stack[-1].num+=1
            while stack:
                if stack[-1].num==2:
                    node=stack.pop()
                else:
                    break
        else:
            parent=stack[-1]
            node=TreeNode(a)
            if parent.num==0:
                parent.left=node
            else:
                parent.right=node
            parent.num+=1
            stack.append(node)
    return node
#def build_tree(s, index):这个美妙多了
    # 如果当前字符为'.'，表示空结点，返回None，并将索引后移一位
    if s[index] == '.':
        return None, index + 1
    # 否则创建一个结点
    node = Node(s[index])
    index += 1
    # 递归构造左子树
    node.left, index = build_tree(s, index)
    # 递归构造右子树
    node.right, index = build_tree(s, index)
    return node, index
def inorder(node):
    if node is None:
        return []
    return inorder(node.left)+[node.val]+inorder(node.right)
def postorder(node):
    if node is None:
        return []
    return postorder(node.left)+postorder(node.right)+[node.val]
s=input().strip()
root=build_tree(s)
l1=inorder(root)
l2=postorder(root)
print(''.join(l1))
print(''.join(l2))
```

括号嵌套树

```
class TreeNode:
    def __init__(self,value):
        self.value=value
        self.children=[]
def build_tree(s):
    stack=[]
```

```
    node=None
    for a in s:
        if a==',':
            continue
        elif a=='(':
            #if node:按规则这里不可能是None因此不需要这个判断
                stack.append(node)
                #node=None应该可以不要
        elif a==')':
            node=stack.pop()#作用应该只是最后可以把根节点赋值上去
        else:
            node=TreeNode(a)
            if stack:#这个应该也只是对一开始根节点有用
                stack[-1].children.append(node)
    return node
#注意：字母用a.isalpha()判断会更安全
```

遍历用邻接表表示的树

```
def main():
    n = int(sys.stdin.readline())
    tree = defaultdict(list)
    all_nodes = set()
    child_nodes = set()

    for _ in range(n):
        parts = list(map(int, sys.stdin.readline().split()))
        parent, *children = parts
        tree[parent].extend(children)
        all_nodes.add(parent)
        all_nodes.update(children)
        child_nodes.update(children)

    # 根节点 = 出现在 all_nodes 但没出现在 child_nodes 的那个
    root = (all_nodes - child_nodes).pop()

    def traverse(u):
        # 把 u 自己和它的所有直接孩子放一起排序
        group = tree[u] + [u]
        group.sort()
        for x in group:
            if x == u:
                print(u)
            else:
                traverse(x)

    traverse(root)
```

当前队列中位数（最小堆，缓删除）

```
from collections import deque
from collections import defaultdict
from heapq import *
n=int(input().strip())
dq=deque()
A=[]
B=[]
to_del=defaultdict(int)
a=0
```

```
b=0

def balance():
    global a,b
    clean()
    if a==b+1:
        heappush(B,-heappop(A))
        a-=1
        b+=1
    if b==a+2:
        heappush(A,-heappop(B))
        b-=1
        a+=1
    clean()


def clean():
    while B and B[0] in to_del:
        to_del[B[0]]-=1
        if to_del[B[0]]==0:
            del to_del[B[0]]
        heappop(B)
    while A and -A[0] in to_del:
        to_del[-A[0]]-=1
        if to_del[-A[0]]==0:
            del to_del[-A[0]]
        heappop(A)

for _ in range(n):
    operation=input().strip()

    if operation=='del':
        num_to_del=dq.popleft()
        to_del[num_to_del]+=1
        if num_to_del>=B[0]:
            b-=1
        else:
            a-=1
        balance()#其实不需要用函数写，反正每次最多偏离1，只要平衡一下即可

    elif operation=='query':
        if b>a:
            print(B[0])
        else:
            m=(B[0]-A[0])/2
            print(int(m) if m.is_integer() else f'{m:.1f}')

    else:
        num=int(operation[4:])
        dq.append(num)
        if b>a:
            heappush(B,num)
            heappush(A,-heappop(B))
            a+=1
        else:
            heappush(A,-num)
            heappush(B,-heappop(A))
            b+=1
        clean()
```

强连通分量（SCC）

```
def kosaraju(n, edges):
    # 构建原图和反图
    graph = [[] for _ in range(n)]
    reverse_graph = [[] for _ in range(n)]

    for u, v in edges:
        graph[u].append(v)
        reverse_graph[v].append(u)

    # 第一次 DFS：记录完成顺序
    visited = [False] * n
    order = []

    def dfs1(u):
        visited[u] = True
        for v in graph[u]:
            if not visited[v]:
                dfs1(v)
        order.append(u)

    for i in range(n):
        if not visited[i]:
            dfs1(i)

    # 第二次 DFS：在反图上找 SCC
    visited = [False] * n
    scc_list = []

    def dfs2(u, component):
        visited[u] = True
        component.append(u)
        for v in reverse_graph[u]:
            if not visited[v]:
                dfs2(v, component)

    # 按完成时间逆序处理
    while order:
        u = order.pop()
        if not visited[u]:
            component = []
            dfs2(u, component)
            scc_list.append(component)

    return scc_list
```

图：

1.最短路径问题

无权：bfs

逃离监狱

```
history=[[[float('inf')]*(K+1) for i in range(C)]for _ in range(R)]
history[sr][sc][0]=0
```

```
from collections import deque
q=deque()
q.append((sr,sc,0))
ans=-1
while q:
    r,c,k=q.popleft()
    n=history[r][c][k]
    if l[r][c]=='E':
        ans=n
        break
    next_steps=[(r+1,c),(r-1,c),(r,c+1),(r,c-1)]
    for nr,nc in next_steps:
        if 0<=nr<R and 0<=nc<C:
            nk=k#一定要另外用一个变量，不能随意改变k，因为k还要继续进入另一个方向的循环
            if l[nr][nc]=='#':
                nk+=1
                if nk>K:
                    continue#此时这一步已经不能走了，不加入队列
            if n+1<history[nr][nc][nk]:
                history[nr][nc][nk]=n+1
                q.append((nr,nc,nk))
```

以及变种（词梯等）有两种构建邻接表的方式（通配符或者直接转成字典搜索）本质一样

```
        for gene in bank:
            for i in range(8):
                pattern=gene[:i]+'*'+gene[i+1:]
                pattern_map[pattern].append(gene)
        graph=defaultdict(list)
        for gene in bank:
            for i in range(8):
                pattern=gene[:i]+'*'+gene[i+1:]
                for neighbor in pattern_map[pattern]:
                    if neighbor!=gene:
                        graph[gene].append(neighbor)
```

有权：

点到点、无负权：dijkstra

```
class Solution:
    def networkDelayTime(self, times: List[List[int]], n: int, k: int) -> int:
        graph=[[inf]*n for _ in range(n)]
        for u,v,w in times:
            graph[u-1][v-1]=w  #建立邻接表
        dist=[inf]*n   #到节点k的距离
        dist[k-1]=0
        done=[False]*n #done[k-1]=True不能要，因为k的条件还没带进去
        ans=0
        while True:
            x=-1
            for i,ok in enumerate(done):
                if not ok and (x<0 or dist[i]<dist[x]):
                    x=i
            if x<0:  #完成
                return ans
            if dist[x]==inf: #已经确定最短距离的点如果还是无穷就是到不了了
                return -1
            ans=dist[x] #最长距离
            done[x]=True
```

```
            for i,d in enumerate(graph[x]):
                dist[i]=min(dist[i],dist[x]+d)
```

点到点、有负权：bellman(只有无负权环的时候才有所谓的最短一说)

```
def bellman_ford(graph, V, source):
    # 初始化距离
    dist = [float('inf')] * V
    dist[source] = 0

    # 松弛 V-1 次
    for _ in range(V - 1):
        for u, v, w in graph:
            if dist[u] != float('inf') and dist[u] + w < dist[v]:
                dist[v] = dist[u] + w

    # 检测负权环
    for u, v, w in graph:
        if dist[u] != float('inf') and dist[u] + w < dist[v]:
            print("图中存在负权环")
            return None

    return dist
```

优化

```
def spfa(graph, V, source):
    """
    使用 SPFA 算法求解单源最短路径。

    参数:
    graph: 邻接表表示的图，graph[u] = [(v, w), ...] 表示从 u 到 v 的边权重为 w
    V: 顶点数量
    source: 源点

    返回:
    dist: 从源点到各顶点的最短距离列表，如果存在负权环则返回 None
    """
    # 初始化距离数组
    dist = [float('inf')] * V
    dist[source] = 0

    # 队列，存储需要处理的顶点
    queue = deque()
    queue.append(source)

    # 入队次数数组，用于检测负权环
    in_queue_count = [0] * V
    in_queue_count[source] = 1

    # 标记顶点是否在队列中，避免重复入队
    in_queue = [False] * V
    in_queue[source] = True

    # SPFA 主循环
    while queue:
        # 取出队首顶点 u
        u = queue.popleft()
        in_queue[u] = False  # 标记为不在队列中
```

```
            # 遍历 u 的所有邻接顶点 v
            for v, w in graph[u]:
                # 尝试松弛边 (u, v)
                if dist[u] != float('inf') and dist[u] + w < dist[v]:
                    dist[v] = dist[u] + w  # 更新最短距离

                    # 如果 v 不在队列中，则将其加入队列
                    if not in_queue[v]:
                        queue.append(v)
                        in_queue[v] = True
                        in_queue_count[v] += 1

                        # 如果某个顶点入队次数超过 V-1 次，则存在负权环
                        if in_queue_count[v] > V - 1:
                            print("图中存在负权环")
                            return None

    return dist
```

所有顶点之间的最短路径：floyd-warshall

```
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
```

2.路径存在问题：dfs（这里以启发式搜索+回溯为例）

```
from  collections import deque
n=int(input())
sr,sc=map(int,input().strip().split())
visited=[[False]*n for _ in range(n)]
visited[sr][sc]=True

def is_valid(r,c):
    if 0<=r<n and 0<=c<n and not visited[r][c]:
        return True
    return False

def next_steps(r,c):
    moves=[[-2,+1],[-2,-1],[-1,+2],[-1,-2],
           [+1,+2],[+1,-2],[+2,+1],[+2,-1]]
    l=[]
    for dr,dc in moves:
        nr,nc=r+dr,c+dc
        if is_valid(nr,nc):
```

```
            count=0
            for ndr,ndc in moves:
                nnr,nnc=nr+ndr,nc+ndc
                if is_valid(nnr,nnc):
                    count+=1
            l.append([nr,nc,count])
    l.sort(key=lambda x:x[2])
    return [[r,c] for r,c,count in l]

def dfs(r,c,number):
    if number==n**2:
        return True
    for nr,nc in next_steps(r,c):
        if is_valid(nr,nc):
            visited[nr][nc]=True
            if dfs(nr,nc,number+1):
                return True
            visited[nr][nc]=False
    return False

if dfs(sr,sc,1):
    print('success')
else:
    print('fail')
```

3.判环

有向图有三种方法：拓扑排序、dfs染色、自创的bfs

拓扑排序

```
from collections import defaultdict,deque
n,m=map(int,input().split())
dic=defaultdict(list)
indegree=[0]*n
for i in range(m):
    u,v=map(int,input().split())
    dic[u].append(v)
    indegree[v]+=1
q=deque()
for i in range(n):
    if indegree[i]==0:
        q.append(i)
count=0
while q:
    node=q.popleft()
    count+=1
    for next_node in dic[node]:
        indegree[next_node]-=1
        if indegree[next_node]==0:
            q.append(next_node)
if count<n:
    print('Yes')
else:
    print('No')
```

dfs染色

```
from collections import defaultdict
dic=defaultdict(list)
```

```
n,m=map(int,input().strip().split())
color=[0]*n
for _ in range(m):
    u,v=map(int,input().strip().split())
    dic[u].append(v)


def dfs(node):
    if color[node]==1:
        return True
    elif color[node]==2:
        return False
    color[node]=1
    for next_node in dic[node]:
        if dfs(next_node):
            return True
    color[node]=2
    return False

judge=0
for i in range(n):
    if color[i]==0:
        if dfs(i):
            print('Yes')
            judge=1
            break
if judge==0:
    print("No")
```

bfs

无向图三种方法：dfs染色、并查集、自创bfs

并查集

4.最小生成树问题

prim算法

```
class Solution:
    def minCostConnectPoints(self, points: List[List[int]]) -> int:
        n=len(points)
        graph=defaultdict(list)
        for i in range(n):
            for j in range(n):
                if i!=j:
                    xi,yi=points[i]
                    xj,yj=points[j]
                    weight=abs(xi-xj)+abs(yi-yj)
                    graph[(i,j)]=weight

        length=[inf]*n
        length[0]=0
        for i in range(1,n):
            length[i]=graph[(0,i)]
```

```
        visited=set([0])
        ans=0

        for _ in range(n-1):
            u=-1
            for v in range(n):
                if v not in visited and (u==-1 or length[v]<length[u]):
                    u=v
            ans+=length[u]
            visited.add(u)
            for i in range(n):
                if i not in visited:
                    length[i]=min(length[i],graph[(u,i)])

        return ans
```

以及一个prim用最小堆的变种(应该适合稀疏)

```
class Solution:
    def minCostConnectPoints(self, points: List[List[int]]) -> int:
        n=len(points)
        graph=defaultdict(list)
        for i in range(n):
            for j in range(n):
                if i!=j:
                    xi,yi=points[i]
                    xj,yj=points[j]
                    weight=abs(xi-xj)+abs(yi-yj)
                    graph[i].append((j,weight))
                    graph[j].append((i,weight))
        edges=[]
        for j,w in graph[0]:
            heapq.heappush(edges,(w,0,j))
        visited = set()#也可以用列表[False]*n
        visited.add(0)
        min_cost=0
        while edges:
            weight,i,j=heappop(edges)
            if j not in visited:
                min_cost+=weight
                visited.add(j)
                for k,w in graph[j]:
                    if k not in visited:
                        heapq.heappush(edges,(w,j,k))
        return min_cost
```

kruskal

```
class Solution:
    def minCostConnectPoints(self, points: List[List[int]]) -> int:
        n=len(points)
        parent=[i for i in range(n)]
        def root(x):
            if parent[x]!=x:
                parent[x]=root(parent[x])#路径压缩
            return parent[x]

        def union(x,y):
            parent[root(x)]=root(y)


        graph=[]
```

```
        for i in range(n):
            for j in range(i,n):
                    xi,yi=points[i]
                    xj,yj=points[j]
                    weight=abs(xi-xj)+abs(yi-yj)
                    heapq.heappush(graph,(weight,i,j))
        ans=0
        count=0
        while graph:
            w,i,j=heapq.heappop(graph)
            if root(i)!=root(j):
                union(i,j)
                count+=1
                ans+=w
            if count==n-1:
                break
        return ans
```

特殊的用自创bfs法更快（0,1）

```
from collections import defaultdict,deque
n,m=map(int,input().strip().split())

graph=defaultdict(set)
for _ in range(m):
    i,j=map(int,input().strip().split())
    i-=1
    j-=1
    graph[i].add(j)
    graph[j].add(i)

unvisited=set(range(n))
def bfs(start):
    q=deque([start])
    while q:
        i=q.popleft()
        to_remove=[]
        for j in unvisited:
            if j!=i and j not in graph[i]:
                q.append(j)
                to_remove.append(j)
        for j in to_remove:
            unvisited.remove(j)
count=0
while unvisited:
    count+=1
    bfs(unvisited.pop())
print(count-1)
```

二分法

```
n,m=map(int,input().split())
l=[]
for i in range(n):
    l.append(int(input().strip()))
i=max(l)
j=sum(l)
while i<=j:
    d=(i+j)//2
    num=1
```

```
    s=0
    for k in range(n):
        if s+l[k]>d:
            s=l[k]
            num+=1
        else:
            s+=l[k]
    if num<=m:
        ans=d
        j=d-1
    else:
        i=d+1
print(ans)
```

动态规划

整数划分

```
# 预处理 dp 表
MAX_N = 50
dp = [[0] * (MAX_N + 1) for _ in range(MAX_N + 1)]

# 初始化：dp[0][m] = 1 对所有 m
for m in range(MAX_N + 1):
    dp[0][m] = 1

# 填表
for n in range(1, MAX_N + 1):
    for m in range(1, MAX_N + 1):
        if m > n:
            dp[n][m] = dp[n][n]
        else:
            dp[n][m] = dp[n][m - 1] + dp[n - m][m]

# 处理输入
import sys
for line in sys.stdin:
    n = int(line.strip())
    print(dp[n][n])
```