cheating sheet

数据结构：

字符串：

“（连接符）”.join(list)

列表：

列表推导式：[声明变量 for 声明变量 in 某集合 if 共同满足的条件]

插入元素：list.insert(index,元素)；bisect库

删除已知元素：list.remove(元素)

删除已知索引的元素：del list[index]

倒序排序：list.sort(reverse=True)

指定顺序排序：list.sort(key= lambda s:排序指标（与s相关）

寻找索引：list.index(元素) 第一个元素，没有会触发ValueError

enumerate()函数（遍历方法：for index，元素代称 in enumerate(列表)）

字典：

{} dict(元组) 半有序：Ordereddict()

遍历字典的键：for 元素 in dict()； for 元素 in dict.keys()

遍历字典的值：for 元素 in dict.values()

删除键值对：del dict[键]

遍历键值对：for key,value in dict.items():

元组：

建立：(...,...,...,...) 含元组的列表：zip(a,b,c,...)

集合：

建立：set()

set.add()一个 set.update()多个

删除元素：set.remove() 或set.discard()（前者有KeyError风险，后者没有）

随机删除：set.pop()

运算：并集：set1 | set2 交集：set1 & set2 差集（补集）：set1 - set2 对称差集（补集之交）：set1^set2

不可变集合：**frozenset()**

库：import

math库

向上取整：math.ceil()

向下取整：math.floor()

阶乘：math.factoria()

数学常数：math.pi（圆周率），math.e（自然对数的底）

math.sqrt(x), math.pow(x,y), math.exp(x), math.log(真数，底数)（默认为自然对数）

math.sin(),math.cos(),math.tan()

math.asin(),math.acos(),math.atan()

heapq库：实现堆

heapq.heapify(list)

heapq.heappush(堆名，被插元素）heapq.heappop(堆名)

插入元素的同时弹出顶部元素：heapq.heappushpop(堆名，被插元素)

（或heapq.heapreplace(堆名，被插元素)）

·以上操作在最大堆中应换为“_X_max”（X是它们中的任意一个）

itertools库：

整数集：itertools.count(x,y)（从x开始往大数的整数，间隔为y）

循环地复制一组变量：itertools.cycle(list)

所有排列：itertools.permutations(集合，选取个数)

所有组合：itertools.combinations

已排序列表去重：[i for i,_ in itertools.groupby(list)]（每种元素只能保留一个）

或者list(group)[:n]（group被定义为分组，保留每组的n个元素）

collections库：

双端队列：

创建：a=deque(list)

从末尾添加元素：a.append(x)

从开头添加元素：a.appendleft(x)

从末尾删除元素：b=a.pop()

从开头删除元素：b=a.popleft()

有序字典：Ordereddict()

默认值字典：a=defaultdict(默认值)，如果键不在字典中，会自动添加值为默认值的键值对，而不报KeyError。

计数器：Counter(str)，返回以字符种类为键，出现个数为值的字典

sys库：

sys.exit()用于及时退出程序

sys.setrecursionlimit()用于调整递归限制（递归层数过多会引起MLE）

statistics库：

1.mean(data)：计算数据的平均值（均值）。

2.harmonic_mean(data)：计算数据的调和平均数。

3.median(data)：计算数据的中位数。

4.median_low(data)：计算数据的低中位数。

5.median_high(data)：计算数据的高中位数。

6.median_grouped(data, interval=1)：计算分组数据的估计中位数。

7.mode(data)：计算数据的众数。

8.pstdev(data)：计算数据的总体标准差。

9.pvariance(data)：计算数据的总体方差。

10.stdev(data)：计算数据的样本标准差。

11.variance(data)：计算数据的样本方差。

数据处理：

二进制：bin()，八进制：oct()，十六进制：hex()

保留n位小数：round(原数字，保留位数)；'%.nf'%原数字；'{:.nf}'.format(原数字)；n位有效数字：'%.ng'%原数字；'{:.ng}'.format(原数字)

ASCII转字符：chr();字符转ASCII：ord()

判断数据类型：isinstance(object,class)

try-except 某error

类的创建：

```
class type(father):
    def __init__(self,specific_level):
        self.character=specific_level
```

算法

dp（动态规划）

步骤：1、定义矩阵（全零或负无穷）

2、遍历矩阵（顺带遍历可选项）

3、遇到可放入：状态转移方程：`a[i][j]=max(a[i-1][j-t]+value[t],a[i-1][j])`,t为物品对空间的占用

4、按情况输出矩阵的一个格子（通常是`a[-1][-1]`）

27401:最佳凑单

- 消费者为了享受商家发布的满减优惠，常常需要面临凑单问题。假设有n件商品，每件的价格分别为p1,p2,...,pn，每件商品最多只能买1件。为了享受优惠，需要凑单价为t。那么我们要找到一种凑单方式，使得凑单价格不小于t（从而享受优惠），同时尽量接近t。被称为“最佳凑单”如果不存在任何一种凑单方式，使得凑单价格不小于t（即无法享受优惠），那么最佳凑单不存在。比如当前还差10元享受满减，可选的小件商品有5件，价格分别为3元、5元、8元、8元和9元，每件商品最多只能买1件。那么当前的最佳凑单就是11元（3元+8元）。
- 输入

  第一行输入商品数n（n <= 100），和需要凑单价t。 第二行输入每件商品的价格。
- 输出

  如果可以凑单成功，则输出最佳凑单的价格 。 如果无法凑单成功，则输出0。

```
n, v=map(int,input().split())#商品数,凑单价
a = list(map(int, input().split())) # 商品价格
sum_a = sum(a)
dp = [[-float("inf")] * (sum_a + 1) for _ in range(n + 1)]
dp[0][0] = 0

for i in range(1,n +1): # 商品数
    for j in range(sum_a+1):#价格,也是重量
        if a[i - 1] <= j:
            dp[i][j] = max(dp[i -1][j], dp[i - 1][j - a[i - 1]] + a[i - 1])
        else:
            dp[i][j] = dp[i -1][j]

if sum_a < v:
    print ("0")
else:
    for k in range(v, sum_a+1):
        if dp[n][k] > 0:
            print(dp[n][k])
            break
```

dfs（深度优先搜索）

步骤：1、定义函数；

2、在函数内部，判断是否终了，若是，存下状态，return；

3、一层层地判断：符合条件就打标记，递归调用，进入下一层；记得在递归调用完后抹除标记，以便搜索其他分支；

4、在函数外调用函数，注意实参是否正确；

5、输出可能结果

八皇后

```
answer = []
def Queen(s):
    for col in range(1, 9):
        for j in range(len(s)):
            if (str(col) == s[j] or # 两个皇后不能在同一列
                    abs(col - int(s[j])) == abs(len(s) - j)): # 两个皇后不能在同一斜
线
                break
        else:
            if len(s) == 7:
                answer.append(s + str(col))
            else:
                Queen(s + str(col))
Queen('')
```

```
def is_safe(board, row, col):
    # 检查同一列是否有皇后
    for i in range(row):
        if board[i] == col:
            return False
    # 检查左上方是否有皇后
    i = row - 1
    j = col - 1
    while i >= 0 and j >= 0:
        if board[i] == j:
            return False
        i -= 1
        j -= 1
    # 检查右上方是否有皇后
    i = row - 1
    j = col + 1
    while i >= 0 and j < 8:
        if board[i] == j:
            return False
        i -= 1
        j += 1
    return True

def queen_dfs(board, row):
    if row == 8:
        # 找到第b个解，将解存储到result列表中
```

```
            ans.append(''.join([str(x+1) for x in board]))
            return
        for col in range(8):
            if is_safe(board, row, col):
                # 当前位置安全，放置皇后
                board[row] = col
                # 继续递归放置下一行的皇后
                queen_dfs(board, row + 1)
                # 回溯，撤销当前位置的皇后
                board[row] = 0
ans = []
queen_dfs([None]*8, 0)
```

bfs（广度优先搜索）

定义好所有方向，按方向遍历就行了。记得标记好走过的地点。

（步骤：1）：from collections import deque；2）：def bfs(start,end):；3）：初始化一个包含起点和步数的双端队列；4）：；5）：；）

寻宝

Billy获得了一张藏宝图，图上标记了普通点（0），藏宝点（1）和陷阱（2）。按照藏宝图，Billy只能上下左右移动，每次移动一格，且途中不能经过陷阱。现在Billy从藏宝图的左上角出发，请问他是否能到达藏宝点？如果能，所需最短步数为多少？

输入

第一行为两个整数m,n，分别表示藏宝图的行数和列数。(m<=50,n<=50)

此后m行，每行n个整数（0，1，2），表示藏宝图的内容。

输出

如果不能到达，输出‘NO’。

如果能到达，输出所需的最短步数（一个整数）。

```
import heapq
def bfs(x,y):
    d=[[-1,0],[1,0],[0,1],[0,-1]]
    queue=[]
    heapq.heappush(queue,[0,x,y])
    check=set()
    check.add((x,y))
    while queue:
        step,x,y=map(int,heapq.heappop(queue))
        if martix[x][y]==1:
            return step
        for i in range(4):
            dx,dy=x+d[i][0],y+d[i][1]
        if martix[dx][dy]!=2 and (dx,dy) not in check:
            heapq.heappush(queue,[step+1,dx,dy])
            check.add((dx,dy))
    return "NO"

m,n=map(int,input().split())
```

```
martix=[list(map(int,input().split())) for i in range(m)]
print(bfs(0,0))
```

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)
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)
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
```

欧拉筛

```
N=20
primes = []
is_prime = [True]*N
is_prime[0] = False;is_prime[1] = False
for i in range(2,N):
    if is_prime[i]:
        primes.append(i)
    for p in primes: #筛掉每个数的素数倍
        if p*i >= N:
            break
        is_prime[p*i] = False
        if i % p == 0: #这样能保证每个数都被它的最小素因数筛掉！
            break
print(primes)
# [2, 3, 5, 7, 11, 13, 17, 19]
```

Tree

```
class TreeNode():
   def __init__(self,val):
        self.value=val
        self.left=None
        self.right=None
N=int(input())
nodes=[TreeNode(i) for i in range(N)]#存放节点，防止重复创建实例
```

```
# 前中建后：
def find_node(a,b):
    if (not a) and (not b):
        return
    root=a[0]
```

```
    for i in range(len(b)):
        if b[i].value==root.value:
            root2=i
    b_left=b[:root2]
    b_right=b[root2+1:]
    a_left=a[1:root2+1]
    a_right=a[root2+1:]
    root.left=find_node(a_left,b_left)
    root.right=find_node(a_right,b_right)
return root
```

```
# 中后建前：
def find_node(a,b):
    if (not a) and (not b):
        return
    root=b[-1]
    for i in range(len(a)):
        if a[i].value==root.value:
            root2=i
            break
    a_left=a[:root2]
    a_right=a[root2+1:]
    b_left=b[:root2]
    b_right=b[root2:len(b)-1]
    root.right=find_node(a_right,b_right)
    root.left=find_node(a_left,b_left)
    return root
```

并查集

步骤：

1. 初始化：将每一个节点的归属设为其自身；
2. 判断：判断两个节点是否同归属，在判断的过程中路径压缩；
3. 合并：将节点的归属按要求重新设置；
4. 关于路径压缩：只要节点的归属不是其父节点，就将其父节点设为总的根节点

```
class Cola():
    def __init__(self,val):
       self.value=val
       self.parent=self
    def find(self):
        if self.parent==self:
            return self
        else:
            self.parent=self.parent.find()
            return self.parent
    def union(self,other):
        root_self = self.find()  # 找到当前对象所在集合的根节点
        root_other = other.find()  # 找到其他对象所在集合的根节点
        if root_self != root_other:  # 如果两个对象不属于同一个集合
            root_other.parent = root_self
            print("No")
```

```
        else:
            print('Yes')
while True:
    try:
        n,m=map(int,input().split())
        colas=[0]+[Cola(i) for i in range(1,n+1)]
        for _ in range(m):
            x,y=map(int,input().split())
            colas[x].union(colas[y])
        counter=set()
        for i in range(1,n+1):
            counter.add(colas[i].find().value)
        counter=list(counter)
        counter.sort()
        counter=list(map(str,counter))
        print(len(counter))
        print(' '.join(counter))
    except EOFError:
        break
```

图

Dijkstra：

```
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])
```

Prim（最小生成树）：

兔子与樱花

```
import heapq
def prim(graph):
    # 初始化最小生成树的顶点集合和边集合
    mst = set()
    edges = []
    visited = set()
    total_weight = 0

    # 随机选择一个起始顶点
    start_vertex = list(graph.keys())[0]
    # 将起始顶点加入最小生成树的顶点集合中
    mst.add(start_vertex)
    visited.add(start_vertex)
    # 将起始顶点的所有边加入边集合中
    for neighbor, weight in graph[start_vertex]:
        heapq.heappush(edges, (weight, start_vertex, neighbor))
    # 循环直到所有顶点都加入最小生成树为止
    while len(mst) < len(graph):
        # 从边集合中选取权重最小的边
        weight, u, v = heapq.heappop(edges)
        # 如果边的目标顶点已经在最小生成树中，则跳过
        if v in visited:
            continue
        # 将目标顶点加入最小生成树的顶点集合中
        mst.add(v)
        visited.add(v)
        total_weight += weight
        # 将目标顶点的所有边加入边集合中
        for neighbor, weight in graph[v]:
            if neighbor not in visited:
                heapq.heappush(edges, (weight, v, neighbor))
    return total_weight

n = int(input())
graph = {}
for _ in range(n - 1):
    alist = list(input().split())
    if alist[0] not in graph.keys():
        graph[alist[0]] = []
    for i in range(1, int(alist[1]) + 1):
        if alist[2 * i] not in graph.keys():
            graph[alist[2 * i]] = []
        graph[alist[0]].append((alist[2 * i], int(alist[2 * i + 1])))
        graph[alist[2 * i]].append((alist[0], int(alist[2 * i + 1])))
print(prim(graph))
```

拓扑排序：

```
from collections import deque, defaultdict
#实际应用中可能需要import heapq
def topological_sort(graph):
    indegree = defaultdict(int)
    result = []
    queue = deque()
    # 计算每个顶点的入度
    for u in graph:
        for v in graph[u]:
            indegree[v] += 1
    # 将入度为 0 的顶点加入队列
    for u in graph:
        if indegree[u] == 0:
            queue.append(u)
    # 执行拓扑排序
    while queue:
        u = queue.popleft()
        result.append(u)
        for v in graph[u]:
            indegree[v] -= 1
            if indegree[v] == 0:
                queue.append(v)
    # 检查是否存在环
    if len(result) == len(graph):
        return result
    else:
        return None
```

单调栈：

```
n=int(input())
lst=list(map(int,input().split()))
stack=[]

for i in range(len(lst)):
    while stack and lst[stack[-1]]<lst[i]:
        lst[stack.pop()]=str(i+1)
    stack.append(i)
while stack:
    lst[stack.pop()]='0'
print(' '.join(lst))#实际操作中，可以再单独开一个数组存放结果，以避免数据覆盖
```

字典树（dict dict型）：

```
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end_of_word = False
class Trie:
    def __init__(self):
        self.root = TrieNode()
```

```
    def insert(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = TrieNode()
                node = node.children[char]
        node.is_end_of_word = True
    def search(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                return False
            node = node.children[char]
        return node.is_end_of_word
    def starts_with(self, prefix):
        node = self.root
        for char in prefix:
            if char not in node.children:
                return False
            node = node.children[char]
        return True
```