计算概论cheat sheet——计概加油!

一．语法：

1. 输入输出类

（1）读取多行输入：

```
法一：应用循环
while True:
    try:
        line = input()  # 读取一行输入
        print(f"你输入的内容是：{line}")#或者主要代码部分
    except EOFError:
        break  # 捕获到 EOFError 时退出循环

举例：快速堆猪（多行输入与辅助栈的应用结合例题）
a = []
m = []
while True:
    try:
        s = input().split()
        if s[0] == "pop":
            if a:
                a.pop()
                if m:
                    m.pop()
        elif s[0] == "min":
            if m:
                print(m[-1])
        else:
            h = int(s[1])
            a.append(h)
            if not m:
                m.append(h)
            else:
                k = m[-1]
                m.append(min(k, h))
    except EOFError:
        break

法二：sys:
import sys# 使用 sys.stdin.read() 读取所有输入
data = sys.stdin.read()  # 读取整个输入流
print(data)
```

(2)list(map(int,input().split()))注意map只是一个生成器，要用list承接，这里应用于输入全是整数

如果不全是整数可以直接input.split() 返回的也是一个列表

注意如果输入中间没有间隔，不要加split，但这样也可以通过`matrix[i][j]`搜索到。

（3）浮点数的输入输出

`float(input())`

```
输出：
法一：f"{value:.nf}"  #其中n是希望保留的小数位数
百分数
print(f"Percentage: {percentage:.2%}")  # 输出：85.00%
科学计数法：
print(f"Scientific notation: {large_number:.2e}")  # 输出：1.23e+06
```

（4）格式化输出：

```
基本模板：f"字符串 {表达式} 字符串"
     print(f"My name is {name} and I am {age} years old.")

三元运算符：
print(f"Adult status: {'Yes' if age >= 18 else 'No'}")
```

（5）其他注意：

```
无空格输出： print(' '.join(map(str, minStep[i]))) #注意这里应该是字符串形式
保护圈： maze.append([-1] + [int(_) for _ in input().split()] + [-1])
```

（6）从列表中删除元素：

```
法一：my_list = [1, 2, 3, 4, 2, 5]
     my_list.remove(2)  # 删除第一个 2
法二：my_list = [1, 2, 3, 4, 5]
     removed_element = my_list.pop(2)  # 删除索引为 2 的元素（即 3）
```

2.字典与集合的使用（包括lambda函数的创建）

（1）字典的使用：

```
# 创建字典
my_dict = {'name': 'Alice', 'age': 25, 'city': 'New York'}

#通过键来搜索值
法一：print(my_dict['name'])  # 输出：Alice
法二：print(my_dict.get('name'))  # 输出：Alice
print(my_dict.get('address', 'Not Found'))  # 输出：Not Found

#通过值来搜索键
找到所有的键：
法一：keys = [key for key, value in my_dict.items() if value == search_value]
法二：keys = list(filter(lambda key: my_dict[key] == search_value, my_dict))
找到第一个符合条件的键：
key = next((key for key, value in my_dict.items() if value == search_value), None)

#添加或更新元素（键值对）：
my_dict['age'] = 26  # 更新
my_dict['country'] = 'USA'  # 添加

#向字典中某一个键下添加元素：
my_dict = {'key1': [1, 2, 3], 'key2': [4, 5]}
my_dict['key1'].append(4)
```

```
#删除键值对
法一：del my_dict['city']     # 删除 'city' 键值对
法二：age = my_dict.pop('age')
print(age)  # 输出：26

#遍历字典：
    # 遍历键
    for key in my_dict:

    # 遍历值
    for value in my_dict.values():
        print(value)

    # 遍历键值对
    for key, value in my_dict.items():
        print(f"{key}: {value}")


#字典推导式举例：
numbers = [1, 2, 3, 4, 5]
squared_dict = {n: n**2 for n in numbers}
print(squared_dict)

#字典排序：
sorted_dict = dict(sorted(my_dict.items(), key=lambda x: x[1], reverse=True))
```

(2) 集合的使用：

```
# 创建集合
my_set = {1, 2, 3, 4, 5}
another_set = set([3, 4, 5, 6, 7])
#注意：set() 用来创建集合时，它接受一个可迭代对象（如列表、元组、字符串等），因而这里set() 会自动从列表中提取元素并创建集合，而不能直接set(3, 4, 5, 6, 7)，因为set()括号里只可以有一个参数，而{}则不同。

# 添加元素
my_set.add(6)
# 删除元素（不存在元素可抛出错误）
my_set.remove(2)
# 删除不存在的元素，不会抛出错误
my_set.discard(10)
```

(3) lambda函数的使用（主要是在字典排序中）

```
基本模板：lambda arguments: expression  #参数：对参数进行的操作

在字典排序中：
sorted_dict = sorted(my_dict.items(), key=lambda x: x[1])
#按值升序排序，注意sorted得到的是一个列表!
#如果想要降序并转化为字典格式如下：
sorted_dict = dict(sorted(my_dict.items(), key=lambda x: x[1], reverse=True))

与map结合：
# 对列表中的每个元素进行平方操作
squared_numbers = list(map(lambda x: x ** 2, numbers))
```

3.排序的几种方法与适用范围（语法方面）：

（1）`sorted()` **返回一个新的排序后的列表**，原始的可迭代对象不被修改。可以作用于任何可迭代对象，包括列表、元组、字典等。支持自定义排序规则（通过 `key` 参数）。用于需要保持原始数据不变的情况，或需要排序结果为列表的场景。

```
如按长度排序：（字典中的用法见上一条）
words = ["banana", "apple", "pear", "cherry"]
sorted_words = sorted(words, key=len)

对列表中的数组按照数组的第一个数字排序：
sorted_dist=sorted(dist,key=lambda x:x[0])
```

（2）`.sort()` 是 **列表对象的方法**，只能对 **列表** 进行排序，原地修改列表，即 排序会直接修改原列表，返回值为 `None`。适用于不需要保留原列表的情况，或者希望节省内存的场景。

4.ASCII表及其用法

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

调用语法：

```
# 获取字符的 ASCII 码——ord()
ascii_value = ord('A')
print(ascii_value)  # 输出：65

# 获取 ASCII 码对应的字符——chr()
char = chr(65)
print(char)  # 输出：A
```

5.enumerate:（返回枚举对象）

基本语法：enumerate(iterable, start=0)
iterable：这是想要遍历的可迭代对象（如列表、元组、字符串等）。
start：可选，指定索引从哪里开始，默认从 0 开始。
返回元组，每个元组由两个元素组成：第一个是索引，第二个是对应的值。
举例1：单纯返回索引和元素

```
fruits = ['apple', 'banana', 'cherry']
for index, fruit in enumerate(fruits):
    print(f"Index: {index}, Fruit: {fruit}")
```

举例2：与if搭配实现查找：

```
fruits = ['apple', 'banana', 'cherry', 'orange']
for index, fruit in enumerate(fruits):
    if 'a' in fruit:
        print(f"Found fruit with 'a' at index {index}: {fruit}")
```

二．算法类：

【一】搜索题

1. bfs模板（最短路径或无权图）：

（1）迷宫最短路径问题

```
from collections import deque

MAX_DIRECTIONS = 4
dx = [0, 0, 1, -1]
dy = [1, -1, 0, 0]

def is_valid_move(x, y, n, m, maze, in_queue):
    return 0 <= x < n and 0 <= y < m and maze[x][y] == 0 and (x, y) not in in_queue

def bfs(start_x, start_y, n, m, maze):
    queue = deque()
    queue.append((start_x, start_y))

    in_queue = set()
    prev = [[(-1, -1)] * m for _ in range(n)]

    in_queue.add((start_x, start_y))
    while queue:
        x, y = queue.popleft()
        if x == n - 1 and y == m - 1:
            return prev
        for i in range(MAX_DIRECTIONS):
            next_x = x + dx[i]
            next_y = y + dy[i]
            if is_valid_move(next_x, next_y, n, m, maze, in_queue):
                prev[next_x][next_y] = (x, y)
                in_queue.add((next_x, next_y))
                queue.append((next_x, next_y))
    return None

def print_path(prev, end_pos):
    path = []
    while end_pos != (-1, -1):
        path.append(end_pos)
        end_pos = prev[end_pos[0]][end_pos[1]]
    path.reverse()
```

```
    for pos in path:
        print(pos[0] + 1, pos[1] + 1)

if __name__ == '__main__':
    n, m = map(int, input().split())
    maze = [list(map(int, input().split())) for _ in range(n)]

    prev = bfs(0, 0, n, m, maze)
    if prev:
        print_path(prev, (n - 1, m - 1))
    else:
        print("No path found")
```

(2) 跨步迷宫

```
from collections import deque

MAXD = 8
dx = [0, 0, 0, 0, 1, -1, 2, -2]
dy = [1, -1, 2, -2, 0, 0, 0, 0]

def canVisit(x, y, n, m, maze, in_queue):
    return 0 <= x < n and 0 <= y < m and maze[x][y] == 0 and (x, y) not in in_queue

def bfs(start_x, start_y, n, m, maze):
    q = deque()
    q.append((0, start_x, start_y))  # (step, x, y)
    in_queue = {(start_x, start_y)}

    while q:
        step, x, y = q.popleft()

        if x == n - 1 and y == m - 1:
            return step

        for i in range(MAXD):
            next_x = x + dx[i]
            next_y = y + dy[i]
            next_half_x = x + dx[i] // 2
            next_half_y = y + dy[i] // 2

            if canVisit(next_x, next_y, n, m, maze, in_queue) and maze[next_half_x]
[next_half_y] == 0:
                in_queue.add((next_x, next_y))
                q.append((step + 1, next_x, next_y))

    return -1

if __name__ == '__main__':
    n, m = map(int, input().split())
    maze = [list(map(int, input().split())) for _ in range(n)]

    step = bfs(0, 0, n, m, maze)
    print(step)
```

(3) 矩阵中的块：

```
from collections import deque
```

```
MAXD = 4
dx = [0, 0, 1, -1]
dy = [1, -1, 0, 0]

def bfs(x, y):
    q = deque([(x, y)])
    inq_set.add((x,y))
    while q:
        front = q.popleft()
        for i in range(MAXD):
            next_x = front[0] + dx[i]
            next_y = front[1] + dy[i]
            if matrix[next_x][next_y] == 1 and (next_x,next_y) not in inq_set:
                inq_set.add((next_x, next_y))
                q.append((next_x, next_y))

n, m = map(int, input().split())
matrix=[[-1]*(m+2)]+[[-1]+list(map(int,input().split()))+[-1] for i in range(n)]+[[-1]*
(m+2)]
inq_set = set()

counter = 0
for i in range(1,n+1):
    for j in range(1,m+1):
        if matrix[i][j] == 1 and (i,j) not in inq_set:
            bfs(i, j)
            counter += 1
print(counter)
```

（4）多终点迷宫问题（现有一个 n*m 大小的迷宫，其中`1`表示不可通过的墙壁，`0`表示平地。每次移动只能向上下左右移动一格，且只能移动到平地上。求从迷宫左上角到迷宫中每个位置的最小步数。输出n行m列个整数，表示从左上角到迷宫中每个位置需要的最小步数。如果无法到达，那么输出`-1`。注意，整数之间用空格隔开，行末不允许有多余的空格。）

```
from collections import deque
import sys

INF = sys.maxsize
MAXD = 4

dx = [0, 0, 1, -1]
dy = [1, -1, 0, 0]

def canVisit(x, y, n, m, maze, in_queue):
    return 0 <= x < n and 0 <= y < m and maze[x][y] == 0 and (x, y) not in in_queue

def BFS(start_x, start_y, n, m, maze):
    minStep = [[-1] * m for _ in range(n)]
    q = deque([(0, start_x, start_y)])  # (step, x, y)
    in_queue = {(start_x, start_y)}
    minStep[start_x][start_y] = 0

    while q:
        step, x, y = q.popleft()

        for i in range(MAXD):
            next_x = x + dx[i]
            next_y = y + dy[i]
            if canVisit(next_x, next_y, n, m, maze, in_queue):
```

```
                in_queue.add((next_x, next_y))
                minStep[next_x][next_y] = step + 1
                q.append((step + 1, next_x, next_y))

    return minStep

n, m = map(int, input().split())
maze = []

for _ in range(n):
    maze.append(list(map(int, input().split())))

minStep = BFS(0, 0, n, m, maze)
for i in range(n):
    print(' '.join(map(str, minStep[i])))    #输出：没有多余空格
```

（5）一维bfs（从整数`1`开始，每轮操作可以选择将上轮结果加`1`或乘`2`。问至少需要多少轮操作才能达到指定整数）：

```
from collections import deque


def bfs(n):
    inq = set()
    inq.add(1)
    q = deque()
    q.append((0, 1))
    while q:
        step, front = q.popleft()
        if front == n:
            return step

        if front * 2 <= n and front * 2 not in inq:
            inq.add(front * 2)
            q.append((step + 1, front * 2))
        if front + 1 <= n and front + 1 not in inq:
            inq.add(front + 1)
            q.append((step + 1, front + 1))


n = int(input())
print(bfs(n))
```

（6）传送门（每次移动只能向上下左右移动一格，且只能移动到平地或传送点上。当位于传送点时，可以选择传送到另一个`2`处（传送不计入步数），也可以选择不传送。求从迷宫左上角到右下角的最小步数。）——注意映射

```
from collections import deque

MAXD = 4

dx = [0, 0, 1, -1]
dy = [1, -1, 0, 0]

def canVisit(x, y, n, m, maze, in_queue):
    return 0 <= x < n and 0 <= y < m and (maze[x][y] == 0 or maze[x][y] == 2) and (x, y)
not in in_queue
```

```
def BFS(start_x, start_y, n, m, maze, transMap):
    q = deque([(0, start_x, start_y)])  # (step, x, y)
    in_queue = {(start_x, start_y)}

    while q:
        step, x, y = q.popleft()

        if x == n - 1 and y == m - 1:
            return step

        for i in range(MAXD):
            next_x = x + dx[i]
            next_y = y + dy[i]
            if canVisit(next_x, next_y, n, m, maze, in_queue):
                in_queue.add((next_x, next_y))
                q.append((step + 1, next_x, next_y))

                if maze[next_x][next_y] == 2:
                    trans_position = transMap.get((next_x, next_y))
                    if trans_position:
                        in_queue.add(trans_position)
                        q.append((step + 1, trans_position[0], trans_position[1]))

    return -1

n, m = map(int, input().split())
maze = []
transMap = {}
transVector = []

for i in range(n):
    row = list(map(int, input().split()))
    maze.append(row)

    if 2 in row:
        for j, val in enumerate(row):
            if val == 2:
                transVector.append((i, j))

        if len(transVector) == 2:
            transMap[transVector[0]] = transVector[1]
            transMap[transVector[1]] = transVector[0]
            transVector = []  # 清空 transVector 以便处理下一对传送点

if transVector:
    print("Error: Unpaired teleportation point found.")
    exit(1)

step = BFS(0, 0, n, m, maze, transMap)
print(step)
```

(7) 变换的迷宫（与双端队列结合）

```
from collections import deque

def bfs(x, y):
    visited = {(0, x, y)}
    dx = [0, 0, 1, -1]
    dy = [1, -1, 0, 0]
    queue = deque([(0, x, y)])
```

```
    while queue:
        time, x, y = queue.popleft()
        for i in range(4):
            nx, ny = x + dx[i], y + dy[i]
            temp = (time + 1) % k
            if 0 <= nx < r and 0 <= ny < c and (temp, nx, ny) not in visited:
                cur = maze[nx][ny]
                if cur == 'E':
                    return time + 1
                elif cur != '#' or temp == 0:
                    queue.append((time + 1, nx, ny))
                    visited.add((temp, nx, ny))
    return 'Oop!'


t = int(input())
for _ in range(t):
    r, c, k = map(int, input().split())
    maze = [list(input()) for _ in range(r)]
    for i in range(r):
        for j in range(c):
            if maze[i][j] == 'S':
                print(bfs(i, j))
```

2. dfs模板

（1）迷宫的可行路径数（从左上角到右下角）

```
MAXN = 5
n, m = map(int, input().split())
maze = []
for _ in range(n):
    row = list(map(int, input().split()))
    maze.append(row)

visited = [[False for _ in range(m)] for _ in range(n)]
counter = 0

MAXD = 4
dx = [0, 0, 1, -1]
dy = [1, -1, 0, 0]

def is_valid(x, y):
    return 0 <= x < n and 0 <= y < m and maze[x][y] == 0 and not visited[x][y]

def DFS(x, y):
    global counter
    if x == n - 1 and y == m - 1:#这里可以改成终点
        counter += 1
        return
    visited[x][y] = True
    for i in range(MAXD):
        nextX = x + dx[i]
        nextY = y + dy[i]
        if is_valid(nextX, nextY):
            DFS(nextX, nextY)
    visited[x][y] = False

DFS(0, 0)#这里可以改为起点，实现任意化
print(counter)
```

（2）指定步数迷宫（问能否从左上角到右下角）

```
MAXN = 5
n, m, k = map(int, input().split())
maze = []
for _ in range(n):
    row = list(map(int, input().split()))
    maze.append(row)

visited = [[False for _ in range(m)] for _ in range(n)]
canReach = False

MAXD = 4
dx = [0, 0, 1, -1]
dy = [1, -1, 0, 0]

def is_valid(x, y):
    return 0 <= x < n and 0 <= y < m and maze[x][y] == 0 and not visited[x][y]

def DFS(x, y, step):#将step放在dfs中实现递归调用
    global canReach
    if canReach:
        return
    if x == n - 1 and y == m - 1:#可改为终点
        if step == k:
            canReach = True
        return
    visited[x][y] = True
    for i in range(MAXD):
        nextX = x + dx[i]
        nextY = y + dy[i]
        if step < k and is_valid(nextX, nextY):
            DFS(nextX, nextY, step + 1)
    visited[x][y] = False

DFS(0, 0, 0)#可改为起点
print("Yes" if canReach else "No")
```

（3）矩阵最大权值路径（注意回溯和列表的拷贝）

```
# 读取输入
n, m = map(int, input().split())
maze = [list(map(int, input().split())) for _ in range(n)]

# 定义方向
directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]  # 右、下、左、上
visited = [[False] * m for _ in range(n)]  # 标记访问
max_path = []
max_sum = -float('inf')  # 最大权值初始化为负无穷

# 深度优先搜索
def dfs(x, y, current_path, current_sum):
    global max_path, max_sum

    # 到达终点，更新结果
    if (x, y) == (n - 1, m - 1):
        if current_sum > max_sum:
            max_sum = current_sum
            max_path = current_path[:]
```

```
        return    #return 后面没有值，表示函数直接结束并返回 None，退出递归调用，回溯的关键。

    # 遍历四个方向
    for dx, dy in directions:
        nx, ny = x + dx, y + dy

        # 检查边界和是否访问过
        if 0 <= nx < n and 0 <= ny < m and not visited[nx][ny]:
            # 标记访问
            visited[nx][ny] = True
            current_path.append((nx, ny))

            # 递归搜索
            dfs(nx, ny, current_path, current_sum + maze[nx][ny])

            # 回溯
            current_path.pop()
            visited[nx][ny] = False

# 初始化起点
visited[0][0] = True
dfs(0, 0, [(0, 0)], maze[0][0])

# 输出结果
for x, y in max_path:
    print(x + 1, y + 1)
```

(4) dfs与dp结合。例题滑雪——反向找到递增

```
r, c = map(int, input().split())
matrix = [list(map(int, input().split())) for _ in range(r)]
dp = [[0 for _ in range(c)] for _ in range(r)]
def dfs(x, y):
    dx = [0, 0, 1, -1]
    dy = [1, -1, 0, 0]
    for i in range(4):
        nx, ny = x + dx[i], y + dy[i]
        if 0 <= nx < r and 0 <= ny < c and matrix[x][y] > matrix[nx][ny]:
            if dp[nx][ny] == 0:
                dfs(nx, ny)
            dp[x][y] = max(dp[x][y], dp[nx][ny] + 1)
    if dp[x][y] == 0:
        dp[x][y] = 1
max_len=0
for i in range(r):
    for j in range(c):
        if not dp[i][j]:
            dfs(i, j)
        max_len = max(max_len, dp[i][j])
print(max_len)
```

3. 迪杰斯特拉算法（最短权值路径）

（1）最短权值路径（现有一个共n个顶点（代表城市）、m条边（代表道路）的无向图（假设顶点编号为从0到n-1），每条边有各自的边权，代表两个城市之间的距离。求从s号城市出发到达t号城市的最短距离。）

```
import heapq
```

```
def dijkstra(n, edges, s, t):
    graph = [[] for _ in range(n)]
    for u, v, w in edges:
        graph[u].append((v, w))
        graph[v].append((u, w))

    pq = [(0, s)]  # (distance, node)
    visited = set()
    distances = [float('inf')] * n
    distances[s] = 0

    while pq:
        dist, node = heapq.heappop(pq)
        if node == t:
            return dist
        if node in visited:
            continue
        visited.add(node)
        for neighbor, weight in graph[node]:
            if neighbor not in visited:
                new_dist = dist + weight
                if new_dist < distances[neighbor]:
                    distances[neighbor] = new_dist
                    heapq.heappush(pq, (new_dist, neighbor))
    return -1
n, m, s, t = map(int, input().split())
edges = [list(map(int, input().split())) for _ in range(m)]
result = dijkstra(n, edges, s, t)
print(result)
```

（2）走山路例题（某同学在一处山地里，地面起伏很大，他想从一个地方走到另一个地方，并且希望能尽量走平路。 现有一个m*n的地形图，图上是数字代表该位置的高度，"#"代表该位置不可以经过。 该同学每一次只能向上下左右移动，每次移动消耗的体力为移动前后该同学所处高度的差的绝对值。现在给出该同学出发的地点和目的地，需要你求出他最少要消耗多少体力。）

```
import heapq
def dfs(start,end,matrix,m,n):
    directions=[(0,1),(1,0),(0,-1),(-1,0)]
    if matrix[start[0]][start[1]]=="#" or matrix[end[0]][end[1]]=="#":
        return "NO"
    choose=[(0,start[0],start[1])]
    record=[[float('inf')for i in range(n)] for j in range(m)]
    record[start[0]][start[1]]=0
    while choose:
        now_effort,x,y=heapq.heappop(choose)
        if (x,y)==end:
            return now_effort
        for i in range(4):
            dx,dy=directions[i]
            nx=x+dx
            ny=y+dy
            if nx>=0 and nx<m and ny>=0 and ny<n and matrix[nx][ny]!="#":
#注意看好mn
                step_effort=abs(int(matrix[nx][ny])-int(matrix[x][y]))
                new_effort=step_effort+now_effort
                if new_effort<record[nx][ny]:#注意这里不要写成小于等于，否则会出现超内存的
问题
                    record[nx][ny]=new_effort
                    heapq.heappush(choose,(new_effort,nx,ny))
```

```
    return 'NO'
m,n,p=map(int,input().split())
matrix=[]
for i in range(m):
    s=input().split()
    matrix.append(s)
for _ in range(p):
    sx,sy,ex,ey=map(int,input().split())
    result=dfs((sx,sy),(ex,ey),matrix,m,n)
    print(result)
```

4. 螺旋矩阵以及对应算法

```
def spiralOrder(matrix):
    rows, columns = len(matrix), len(matrix[0])
    visited = [[False] * columns for i in range(rows)]
    total = rows * columns
    result = []

    directions = [[0, 1], [1, 0], [0, -1], [-1, 0]]
    row, column = 0, 0
    directionIndex = 0

    for i in range(total):
        result.append(matrix[row][column])
        visited[row][column] = True

        nextRow, nextColumn = row + directions[directionIndex][0], column +
directions[directionIndex][1]

        if not (0 <= nextRow < rows and 0 <= nextColumn < columns and not
visited[nextRow][nextColumn]):
            directionIndex = (directionIndex + 1) % 4


        row += directions[directionIndex][0]
        column += directions[directionIndex][1]

    return result
```

【二】动态规划（包括双dp），模板找拦截导弹，背包问题

（1）双dp（土豪购物）——两种情况分析，取或不取，放或不放

```
def max_value(s):
    n=len(s)
    dp1=[0 for _ in range(n)]
    dp2=[0 for _ in range(n)]
    dp1[0]=s[0]
    dp2[0]=s[0]
    for i in range(1,n):
        dp1[i]=max(dp1[i-1]+s[i],s[i])#不放回
        dp2[i]=max(dp2[i-1]+s[i],dp1[i-1],s[i])#放回之前某个，放回现在也就是第i个，从现在开始取
    return max(max(dp1),max(dp2))

s=list(map(int,input().split(',')))
max_num=max_value(s)
```

```
print(max_num)
```

类似思路：最大摆动子序列

```
def max_len(n,nums):
    if n==1:
        return 1
    else:
        up=1
        down=1
        for i in range(1,n):
            if nums[i]>nums[i-1]:
                up=down+1
            elif nums[i]<nums[i-1]:
                down=up+1
    return max(up,down)

n=int(input())
nums=list(map(int,input().split()))
print(max_len(n,nums))
```

（2）拦截导弹

```
def max_intercepted_missiles(k, heights):
    # Initialize the dp array
    dp = [1] * k

    for i in range(1, k):
        for j in range(i):
            if heights[i] <= heights[j]:
                dp[i] = max(dp[i], dp[j] + 1)

    return max(dp)

if __name__ == "__main__":

    k = int(input())
    heights = list(map(int, input().split()))

    result = max_intercepted_missiles(k, heights)
    print(result)
```

（3）背包问题（小偷背包——01背包取或不取）

第一行是两个整数N和B，空格分隔。N表示物品件数，B表示背包最大承重。 第二行是N个整数，空格分隔。表示各个物品价格。 第三行是N个整数，空格分隔。表示各个物品重量（是与第二行物品对齐的）。

输出一个整数。保证在满足背包容量的情况下，偷的价值最高。

```
n,b=map(int, input().split())
price=[int(i) for i in input().split()]   #（注意这里下标从零开始）
weight=[int(i) for i in input().split()]
bag=[[0]*(b+1) for _ in range(n+1)]
for i in range(1,n+1):
    for j in range(1,b+1):
        if weight[i]<=j:
            bag[i][j]=max(price[i]+bag[i-1][j-weight[i]], bag[i-1][j])
        else:
            bag[i][j]=bag[i-1][j]
print(bag[-1][-1])
```

（4）完全背包（无数个）

```
n, a, b, c = map(int, input().split())
dp = [float('-inf')]*n
for i in range(1, n+1):
    for j in (a, b, c):
        if i >= j:
            dp[i] = max(dp[i-j] + 1, dp[i])
print(dp[n])
```

（5）多重背包（每个物品数量有上线）：最简单的思路是将多个同样的物品看成多个不同的物品，从而化为0-1背包。稍作优化：可以改善拆分方式，譬如将m个1拆成x_1,x_2,……,x_t个1，只需要这些x_i中取若干个的和能组合出1至m即可。最高效的拆分方式是尽可能拆成2的幂，也就是所谓“二进制优化”

【三】贪心算法（重在想明白）

【四】内置函数以及其调用方法

1. Heapq

   `heapq.heappush(heap, item)`

   - 将元素 `item` 推入堆中，并保持堆的性质。

   `heapq.heappop(heap)`

   - 弹出并返回堆中的最小元素，同时保持堆的性质。

   `heapq.heappushpop(heap, item)`

   - 将元素 `item` 推入堆中，并弹出并返回堆中的最小元素。这是一个优化方法，执行这两步比单独调用 `heappush` 和 `heappop` 更高效。

   举例，毒药：类似反悔写法

   ```
   import heapq
   def max_value(n,lst):
       health=0
       choose=[]
       for i in range(n):
           if health+lst[i]>=0:
               health+=lst[i]
               heapq.heappush(choose,lst[i])#有列表和元素
           else:
               if not choose:
                   continue
               if lst[i]>choose[0]:
                   health-=heapq.heappop(choose)
                   heapq.heappush(choose,lst[i])
                   health+=lst[i]
       return len(choose)
   ```

```
n=int(input())
lst=list(map(int,input().split()))
max_num=max_value(n,lst)
print(max_num)
```

2. deque，queue

（1）`deque` — 双端队列

`append(x)`：将元素 `x` 添加到右端。

`appendleft(x)`：将元素 `x` 添加到左端。

`pop()`：从右端弹出元素。

`popleft()`：从左端弹出元素。

`extend(iterable)`：在右端添加多个元素。

`extendleft(iterable)`：在左端添加多个元素（注意，它是逆序添加的）。

```
from collections import deque
dq = deque()
dq.append(1)
dq.pop()
dq.extend([4, 5])
```

（2）`queue` — 队列模块

`put(item)`：将元素 `item` 放入队列（如果队列已满，会阻塞）。

`get()`：从队列取出元素（如果队列为空，会阻塞）。

`empty()`：判断队列是否为空。

`full()`：判断队列是否满。

`qsize()`：返回队列的大小。

（多与bfs结合）

3. bisect:用于在已排序的列表中找到插入点，或者对列表进行插入操作，同时保持列表的顺序。常见的应用包括二分查找（binary search）、在排序列表中插入元素等。

```
1.bisect_left(arr, x):（bisect_right(arr, x)同理）
返回一个索引，该索引是列表 arr 中插入元素 x 的位置，并且会确保 x 插入后，列表仍然保持升序排列。
如果 x 已经存在于列表中，则返回左边的插入位置（即 x 的第一个位置）。

import bisect
arr = [1, 3, 4, 10, 12]
index = bisect.bisect_left(arr, 5)
print(index)  # 输出 3，因为 5 应该插入在 10 之前，索引 3

2.insort(arr, x):（insort_left(arr,x) 保持升序，若x已经存在，插入左边，
insort_right(arr,x)同理）
将元素 x 插入到列表 arr 中，并保持列表的顺序。
等效于使用 bisect_right 找到插入位置，然后插入元素。

import bisect
arr = [1, 3, 4, 10, 12]
bisect.insort(arr, 5)  # 将 5 插入到正确的位置
print(arr)  # 输出 [1, 3, 4, 5, 10, 12]
```

二分查找实现：

```
import bisect
arr = [1, 3, 4, 10, 12]
x = 4
pos = bisect.bisect_left(arr, x)
if pos < len(arr) and arr[pos] == x:
    print(f"元素 {x} 存在于列表中，位置为 {pos}")
else:
    print(f"元素 {x} 不在列表中")
```

区间查找：

```
import bisect
intervals = [1, 5, 10, 15, 20]  # 代表的区间为 [1, 5), [5, 10), [10, 15), [15, 20)
value = 12
index = bisect.bisect_right(intervals, value)
print(f"数值 {value} 在区间 {intervals[index-1]} 和 {intervals[index]} 之间")
```

【五】回溯

回溯的基本步骤如下：

1. **选择**：在当前状态下选择一个可行的选项。
2. **探索**：基于选择，递归地进入下一层状态，继续进行选择。
3. **撤销选择**：如果当前选择不再满足问题的约束，或者搜索到问题的边界，就回溯，撤销当前的选择，返回上一步，尝试其他的选择。

八皇后问题：（会下国际象棋的人都很清楚：皇后可以在横、竖、斜线上不限步数地吃掉其他棋子。如何将8个皇后放在棋盘上（有8 * 8个方格），使它们谁也不能被吃掉！这就是著名的八皇后问题。 对于某个满足要求的8皇后的摆放方法，定义一个皇后串a与之对应，即$a=b_1b_2...b_8~$,其中b_i为相应摆法中第i行皇后所处的列数。已经知道8皇后问题一共有92组解（即92个不同的皇后串）。 给出一个数b，要求输出第b个串。串的比较是这样的：皇后串x置于皇后串y之前，当且仅当将x视为整数时比y小。）

```
list1 = []

def queen(s):
    if len(s) == 8:
        list1.append(s)
        return
    for i in range(1, 9):
        if all(str(i) != s[j] and abs(len(s) - j) != abs(i - int(s[j])) for j in
range(len(s))):
            queen(s + str(i))

queen('')
samples = int(input())
for k in range(samples):
    print(list1[int(input()) - 1])
```

全排列问题：（现在给你一个正整数n，n小于8，输出数组[1, 2, ...，n]的从小到大的全排列。

输入

输入有多行，每行一个整数。当输入0时结束输入。

输出

对于每组输入，输出该组的全排列。每一行是一种可能的排列，共n个整数，每个整数用一个空格隔开，每行末尾没有空格。

```
def dfs(n, path, used, res):
    if len(path) == n:
        res.append(path[:])
        return
    for i in range(1, n+1):
        if not used[i]:
            used[i] = True
            path.append(i)
            dfs(n, path, used, res)
            path.pop()
            used[i] = False

def print_permutations(n):
    res = []
    dfs(n, [], [False]*(n+1), res)
    for perm in sorted(res):
        print(' '.join(map(str, perm)))

nums = []
while True:
    num = int(input())
    if num == 0:
        break
    nums.append(num)

for num in nums:
    print_permutations(num)
```

方法二：引入内置函数：

```
perms = itertools.permutations(arr, 2(c这里是长度))

for perm in perms:
    print(perm)itertools.permutations(iterable, r=None)
```

【六】排序（包括双指针双重循环）归并排序中的递归

```
#冒泡排序
for i in range(n):
    for j in range(n-1-i):
        if l[j] + l[j+1] > l[j+1] + l[j]:
            l[j],l[j+1] = l[j+1],l[j]
```

【七】递归（在题目中已经体现，最小问题）

```
# 读取输入
N = int(input())  # 单词数量
blocks = [set(input()) for _ in range(4)]  # 每个积木上的字母（4个积木，每个积木6个字母）
words = [input().strip() for _ in range(N)]  # 单词列表

# 判断是否能够拼出某个单词的回溯函数
def can_spell(word, used_blocks):
    if len(word) == 0:
        return True  # 如果单词为空，说明拼写成功
```

```
    current_letter = word[0]  # 当前需要拼写的字母
    for i in range(4):  # 检查四块积木
        if current_letter in blocks[i] and i not in used_blocks:  # 如果当前积木面有字母且该积木未使用
            used_blocks.add(i)  # 标记该积木已使用
            if can_spell(word[1:], used_blocks):  # 递归拼写剩余的字母
                return True  # 如果拼写成功，返回 True
            used_blocks.remove(i)  # 回溯，撤销当前选择
    return False  # 如果无法拼出字母，返回 False

# 对每个单词进行判断
def solve():
    for word in words:
        if can_spell(word, set()):
            print("YES")
        else:
            print("NO")

solve()
```

【八】双指针：

```
def binary_search(arr, target):
    left, right = 0, len(arr) - 1

    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid  # 返回目标元素的索引
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1  # 如果未找到目标元素，返回 -1
```

【九】其他小技巧（剪枝，打标记，滑动窗口）

1.滑动窗口（本质可以从双指针出发）：

初始化：

- 维护一个窗口 `[start + 1, i]`，表示当前的无重复子串。
- 使用一个字典 `char_index` 来记录每个字符最近一次出现的位置。

扩展窗口：

- 遍历字符串，逐个字符地扩展窗口的右边界 `i`。

收缩窗口：

- 如果当前字符 `c` 在字典中且其上次出现的位置在当前窗口内，则需要收缩窗口的左边界 `start`，使其不包含重复字符。

模板：

```
    def lengthOfLongestSubstring(s):
        start = -1  # 当前无重复子串的起始位置的前一个位置
        max_length = 0  # 最长无重复子串的长度
        char_index = {}  # 字典，记录每个字符最近一次出现的位置
        for i, char in enumerate(s):#遍历
            if char in char_index and char_index[char] > start:   # 如果字符在字典中且上次出现的位置大于当前无重复子串的起始位置
                start = char_index[char] # 更新起始位置为该字符上次出现的位置
            char_index[char] = i         # 更新字典中字符的位置
            current_length = i - start # 计算当前无重复子串的长度
            max_length = max(max_length, current_length)

        return max_length
```

2.字典序的大小比较

```
n = int(input())
num=input.split()
for i in range(n-1):
    for j in range(i+1,n):
        if num[i]+num[j]<num[j]+num[i]:--这里是字符串的组合，比较的是字典序
            num[i],num[j]=num[j],num[i]
ans = ''.join(num)
num.reverse()
ans2 = ''.join(num)
print(ans+' ' +''.join(ans2))
```

3.stack(包括辅助栈)：后进先出

1. **入栈 (Push)**: 使用`append()`方法将元素压入栈。
2. **出栈 (Pop)**: 使用`pop()`方法将栈顶元素弹出。
3. **栈顶元素 (Peek)**: 使用索引`[-1]`访问栈顶元素。
4. **检查栈是否为空**: 使用`if not stack`来判断栈是否为空。

```
stack.append(1)  # 栈: [1]
print("栈顶元素:", stack[-1])
top = stack.pop()
print("出栈元素:", top)
print("栈是否为空:", len(stack) == 0)  # 输出: False
stack.clear()  # 栈变为空: []
print("栈是否为空:", len(stack) == 0)  # 输出: True
```

例题：括号匹配：

```
def is_valid_parentheses(s)
    stack = []
    # 括号映射
    parentheses_map = {')': '(', '}': '{', ']': '['}
    for char in s:
        if char in parentheses_map.values():
            stack.append(char)
        elif char in parentheses_map.keys():
            if not stack or stack.pop() != parentheses_map[char]:
                return False
    return not stack
```

```
# 测试
print(is_valid_parentheses("()[]{}"))  # 输出: True
```

4.区间问题：

按照右端点排序：

（1）不相交区间数目最大

（2）区间选点问题（给出一堆区间，取**尽量少**的点，使得每个区间内**至少有一个点**）——尽量选择当前区间最右边的点，同不相交区间数目最大

(3)区间覆盖问题：给出一堆区间和一个目标区间，问最少选择多少区间可以**覆盖**掉题中给出的这段目标区间。

按照左端点排序：

（1）区间合并（左端点由小到大排序，维护前面区间右端点ed）

（2）区间分组问题：从**前往后**依次枚举每个区间，判断当前区间能否被放到某个现有组里面。

三.其他易错点：

1.注意矩阵问题中m,n是行数还是行索引，考虑

```
（1）for i in range(n):
        for i in range(m):这一类创建出来的列表行列是否与题意符合
（2）if 0<=nx<n and 0<=ny<m 看好判断条件
```

2.校门外的树，区间中的整数问题（找到总共给定范围内被去除的树的数量，注意用range和集合搭配可以实现对重叠整数的去除，实现不重不漏。如果像最开始那样考虑范围的重叠性再考虑边界和内部的整点不仅麻烦而且不易于实现用时较长。）

```
L,M=map(int,input().split())
regions=[]
for i in range(M):
    a,b=map(int,input().split())
    regions.append((min(a,b),max(a,b)))
regions_set=set()
for start,end in regions:
    regions_set.update(range(start,end+1))
remaining_trees=L+1-len(regions_set)
print(remaining_trees)
```

3.求解素数的三种方法

```
朴素法（时间复杂度是O(N^2)）
primesNumber = []
def is_prime(n):
    for i in range(2, n-1):
        if n % i == 0:
            return False
    return True
def primes(number):
    for i in range(2, number):
        if is_prime(i):
            primesNumber.append(i)
primes(10000)
print(primesNumber)
```

埃氏筛

```
def sieve_of_eratosthenes(n):
    # 创建一个布尔列表，初始化为 True，表示所有数字都假设为素数
    primes = [True] * (n + 1)
    primes[0] = primes[1] = False  # 0 和 1 不是素数

    # 从 2 开始，处理每个数字
    for i in range(2, int(n**0.5) + 1):
        if primes[i]:  # 如果 i 是素数
            # 将 i 的所有倍数标记为非素数
            for j in range(i * i, n + 1, i):
                primes[j] = False

    # 返回所有素数
    return [x for x in range(2, n + 1) if primes[x]]
```

欧拉筛

```
# 返回小于r的素数列表
def oula(r):
    # 全部初始化为0
    prime = [0 for i in range(r+1)]
    # 存放素数
    common = []
    for i in range(2, r+1):
        if prime[i] == 0:
            common.append(i)
        for j in common:
            if i*j > r:
                break
            prime[i*j] = 1
            #将重复筛选剔除
            if i % j == 0:
                break
    return common
prime = oula(20000)
print(prime)
```

4.浅拷贝与深拷贝问题：

```
浅拷贝：
import copy
shallow_copy = copy.copy(original)
深拷贝：
import copy
deep_copy = copy.deepcopy(original)
```

注：二维数组的浅拷贝问题：（1）不能用[:]会发生浅拷贝

（2）创建列表时不要两个*浅拷贝

浅拷贝：对于 **非嵌套的一维数组**（列表），`[:]` 创建的新列表是完全独立的，修改新列表的元素不会影响原列表。但对于 **嵌套数组**（例如，列表中的列表），它们共享嵌套对象的引用，修改内部嵌套对象会影响两个列表。