目录：

输入

- 输入 k 行的矩阵：**matrix = [list(map(int, input().split())) for _ in range(k)]**
- 输入一个 n*n 的矩阵并且把每一个初始值都赋值为 0：**D = [[0]*n for _ in range(n)]**
- 加保护圈的矩阵：**（n 行 m 列）**

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

或者这样：

```
t=[[0 for _ in range(m+2)] for _ in range(n+2)]
for i in range(1,n+1):
    t[i][1:-1]=map(int,input().split())
```

- 一次性读取所有输入：

```
import sys
input = sys.stdin.read
data = input().split() # 读入所有数据并分割为列表
```

一次性输出所有数据：（代替 print）**sys.stdout.write('\n'.join(map(str, results)) + '\n')**

- 没有明确指示停止的输入：

```
while True:
    try:
        a,b=input().split()
    except EOFError:
        Break
```

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'}")**
- 保留小数输出：'%.2f'%__（会四舍五入，什么不会四舍五入？）：例：print('%.2f'%3.1415926)=3.14；**print(f"{value:.2f}")**；百分数：**print(f"Percentage: {value:.2%}")**
- 字符串输出：**print("\n ".join(map(str,new_list)))** **\n** 是换行；先把列表里的元素转换为字符串

- 列表式输出：**list(map(print, my_list))**
- 输出矩阵：

  for i in range(1,n+1): 这是在有保护圈的情况下

  print(' '.join(map(str,matrix[i][1:-1])))（注：str 的作用是先化为字符串）

生成器表达式：

```
squared_gen = (x * x for x in range(5))
for num in squared_gen:
    print(num)
```

与 for 循环显式循环对比：

```
squared_list = []
for x in range(5):
    squared_list.append(x * x)
for num in squared_list:
    print(num)
```

列表推导式：用于基于现有的可迭代对象创建新的列表。通常用来替代传统的 for 循环

例：**even_squares = [x * x for x in range(10) if x % 2 == 0]**

也可在列表推导式中应用多个 for 循环：**pairs = [(x, y) for x in range(3) for y in range(3)]**

字典推导式：**numbers = [1, 2, 3, 4, 5]；squared_dict = {n: n**2 for n in numbers};**

字符串：

- 大小写转换：**str.upper(); str.lower(); str.swapcase()**大小互换; **str.capitalize()**首字母大写; **str.title()**单词首字母大写
- **str.replace(old, new, count) count** 的作用是只替换前 count 次出现的 old
- **str.count(substring, start, end)**
- **str.startswith('xxxx')** 用于判断字符串是否以指定的前缀开头，返回值为 True 或 False。
- **str.strip('xxxx')**；移除左边为 lstrip(); 右边为 rstrip()

列表：

- **enumerate** 用法：

  ```
  for idx, value in enumerate(my_list, start=1):
      print(idx, value)
  ```

  用 enumerate 创建字典：**word_dict={idx: word for idx, word in enumerate(my_list)}**

  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}")
  ```

- **sort:**

  sorted_T = sorted(T, key=lambda x: (x[0], x[1]))（先根据第一个元素排，再根据第二个）

  按长度排序：**sorted_words = sorted(words, key=len)**
- **my_list.remove(2) #** 删除第一个元素 2
- **removed_element = my_list.pop(2) #** 删除索引为 2 的元素
- **lambda** 用法：**squared_numbers = list(map(lambda x: x ** 2, numbers))**

Python包含以下方法:

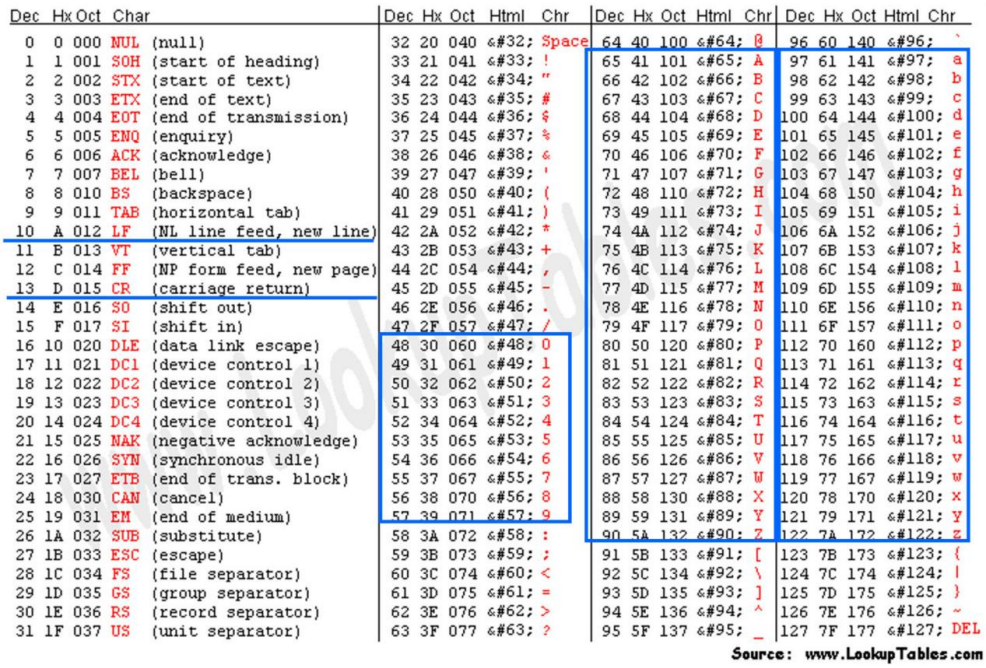
序号	方法
1	list.append(obj) 在列表末尾添加新的对象
2	list.count(obj) 统计某个元素在列表中出现的次数
3	list.extend(seq) 在列表末尾一次性追加另一个序列中的多个值（用新列表扩展原来的列表）
4	list.index(obj) 从列表中找出某个值第一个匹配项的索引位置
5	list.insert(index, obj) 将对象插入列表
6	list.pop([index=-1]) 移除列表中的一个元素（默认最后一个元素），并且返回该元素的值
7	list.remove(obj) 移除列表中某个值的第一个匹配项
8	list.reverse() 反向列表中元素
9	list.sort(cmp=None, key=None, reverse=False) 对原列表进行排序

字典：

- 创建字典：**my_dict = {'name': 'Alice', 'age': 25, 'city': 'New York'}**
- 通过键来搜索值：**print(my_dict['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：**keys = [key for key, value in my_dict.items() if value == search_value]**
  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 ； my_dict['country'] = 'USA'**
- 向字典中某一个键下添加元素：
  my_dict = {'key1': [1, 2, 3], 'key2': [4, 5]}；添加：**my_dict['key1'].append(4)**
- 删除键值对：**del my_dict['city'] #** 删除 'city' 键值对
  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}")**
- 字典排序：**sorted_dict = dict(sorted(my_dict.items(), key=lambda x: x[1], reverse=True))**
  (sorted 得到的只是一个列表，想要转化为字典则需要在外面加上 dict)

集合：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）
my_set = set([3, 4, 5, 6, 7])：或 **set={3,4,5,6,7}**;
my_set.add(6); my_set.remove(2); my_set.discard(10)（若元素不在不会报错）

ASCII 表：



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

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

包：

- **math.ceil()（向上取整）； math.floor()（向下取整）； math.sqrt()（开平方根）； math.trunc()（保留整数部分）**
- **Itertools: (import itertools)**对数据进行排列/组合/重复/筛选；

  A=itertools.count(start=0, step=1):生成一个无限序列，从 start 开始， step 为步长递增

  A=itertools.chain(*iterables):将多个可迭代对象连接在一起，形成一个单一的迭代器。

  ch = itertools.chain([1, 2, 3], ['a', 'b', 'c'])

  print(list(ch)) (结果： [1, 2, 3, 'a', 'b', 'c'])

  A=itertools.permutations(iterable, r=None):返回输入可迭代对象的所有 r 长度的排列

  perm = itertools.permutations([1, 2, 3], 2)

  print(list(perm)) (结果： [(1, 2), (1, 3), (2, 1), (2, 3), (3, 1), (3, 2)])

  A=itertools.combinations(iterable, r):返回输入可迭代对象的所有 r 长度的组合

  comb = itertools.combinations([1, 2, 3], 2)

  print(list(comb)) (结果： [(1, 2), (1, 3), (2, 3)])

  A=itertools.combinations_with_replacement(iterable, r):返回带有重复元素的所有 r 长度的组合（允许重复元素）（例如，有(1,1), (2,2), (3,3)）

  其它：**from itertools import repeat; a=repeat(obj, time)**

  itertools.filterfalse(lambda x: x%2==0, range(10))（过滤满足条件元素）；
- deque — 双端队列：**a=deque()**

  a.append(x) ： 将元素 x 添加到右端；**a.appendleft(x)** ： 将元素 x 添加到左端。

  a.pop() ： 从右端弹出元素；**a.popleft()** ： 从左端弹出元素。

  a.extend(iterable) ： 在右端添加多个元素；**a.extendleft(iterable)** ： 在左端添加多个元素
- **defaultdict: (from collections import defaultdict)** 当访问一个不存在的键时，defaultdict 会自动为这个键创建一个默认值

  d=defaultdict(int); d['a']返回默认值 0 ; d=defaultdict(list);d['a']返回默认空列表[]; d=defaultdict(set); d['a']返回空集合 set()

  常用来进行元素计数：

  word_count = defaultdict(int)

  words = ['apple', 'banana', 'apple', 'orange', 'banana', 'apple']

```
for word in words:
    word_count[word] += 1
print(word_count)# defaultdict(<class 'int'>, {'apple': 3, 'banana': 2, 'orange': 1})
```

（或 **print(dict(word_count))**，则只生成字典）

- **heapq: import heapq**

  heapq.heappush(heap, item): 压进堆的元素会自动从小到大排列

  heapq.heappop(heap)：弹出并返回堆中最小的元素，同时保持堆的性质。

  heapq.heapify(my_list)把列表转为堆

  heapq.heappushpop(heap, item)：将 item 插入堆后再弹出最小值，结合了 heappush 和 heappop

  如果需要实现**最大堆**，可以通过取负值实现：

```
data = [1, 3, 5, 7, 9]
max_heap = []
for item in data:
    heapq.heappush(max_heap, -item)  # 插入负值
print(-heapq.heappop(max_heap))  # 9，弹出最大值
```

- 用 dfs 做 dp 类的题，只要用这个就不会超时：

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

- 拷贝：import copy

  浅拷贝：**shallow_copy = copy.copy(original)**

  深拷贝：**deep_copy = copy.deepcopy(original)**

算法类：

（一）搜索题

dfs 模板题：

迷宫的可行路径数：

```
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)
```

最大连通域面积：

```
directions=[(1,0),(-1,0),(1,1),(-1,1),(0,1),(0,-1),(-1,-1),(1,-1)]
def dfs(x,y):
    cnt = 1
    maze[x][y]='.'
    for dx,dy in directions:
        nx=x+dx
        ny=y+dy
        if 0<=nx<N and 0<=ny<M and maze[nx][ny]=='W':
            cnt += dfs(nx,ny)
    return cnt
T=int(input())
for _ in range(T):
    N,M=map(int,input().split())
    maze=[list(input()) for _ in range(N)]
    max_area=0
    for i in range(N):
        for j in range(M):
            if maze[i][j]=='W':
                max_area=max(max_area,dfs(i,j))
    print(max_area)
```

指定步数迷宫（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")
```

矩阵最大权值（求路径与最大值，有回溯和 visited 的版本）

```
directions=[(1,0),(-1,0),(0,1),(0,-1)]
n,m=map(int,input().split())
maze=[list(map(int,input().split())) for _ in range(n)]
visited=[[False]*m for _ in range(n)]
max_sum=-float('inf')
max_path=[]
def dfs(x,y,cur_sum,cur_path):
    global max_sum
    global max_path
    if x==n-1 and y==m-1:（终止条件）
        if cur_sum>max_sum:
            max_sum=cur_sum
            max_path=cur_path[:]
        return
    for dx,dy in directions:
        nx=x+dx
        ny=y+dy
        if 0<=nx<n and 0<=ny<m and visited[nx][ny]==False:
            visited[nx][ny]=True
            cur_path.append((nx,ny))
            cur_sum+=maze[nx][ny]

            dfs(nx,ny,cur_sum,cur_path)
            （下面三行都是回溯）
            cur_sum-=maze[nx][ny]
            cur_path.pop()
            visited[nx][ny]=False
visited[0][0]=True
dfs(0,0,maze[0][0],[(0,0)])
for x,y in max_path:
    print(x+1,y+1)
```

迪杰斯特拉算法（最短路径；bfs+greedy）：

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)
```

bfs 模板题：

OJ19930：寻宝代码

```
from collections import deque
dx = [0, 0, 1, -1]
dy = [1, -1, 0, 0]
def bfs(x,y):
    q=deque()
    q.append((x,y))
    inq = set()
    inq.add((x,y))
    step=0
    while q:
        for _ in range(len(q)):
            cur_x, cur_y = q.popleft()
            if maze[cur_x][cur_y]==1:
                return step
            for direction in range(4):
                nx = cur_x + dx[direction]
                ny = cur_y + dy[direction]
                if 1<=nx<=m and 1<=ny<=n and maze[nx][ny]!=2 and (nx,ny) not in inq:
                    inq.add((nx,ny))
                    q.append((nx,ny))
        step += 1
    return 'NO'

m, n = map(int, input().split())
maze = [[-1] * (n + 2)] + [[-1] + list(map(int, input().split())) + [-1] for i in range(m)] + [[-1] * (n + 2)]（m 是行,
```

n 是列）

print(bfs(1,1))

迷宫最短路径：

```
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")
```

矩阵中的块：

```
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)
```

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
```

螺旋矩阵：

```
n=int(input())
mx = [[401 for _ in range(n+2)]for _ in range(n+2)]
（因为 1<=n<=20，所以 n*n<=400；以及套保护圈）
for i in range(1,n+1):
    mx[i][1:-1]=[0]*n （把中间要赋值的那部分都赋为 0）
directions=[[0,1],[1,0],[0,-1],[-1,0]]（排序是按向右向下向左向上来排的）
row=1 第一行
col=1 第一列
N=0
d_row,d_col=directions[0]
for j in range(1,n*n+1): （遍历 1 到 n*n）
    mx[row][col]=j
    if mx[row+d_row][col+d_col]!=0:（碰到保护圈了）
        N+=1
        d_row, d_col = directions[N%4] （换方向）
    row+=d_row
    col+=d_col
for i in range(1,n+1):
```

```
    print(' '.join(map(str,mx[i][1:-1])))（输出矩阵的方式）
```
（注：for row in square: 和 for value in row: 主要用于逐行逐元素地遍历二维矩阵的所有元素，是处理矩阵问题中最常见的遍历方式之一。For row in square; print(row)将矩阵的每一行单独作为一个列表输出；for row in square + for value in row + print(value, end=””) 输出该矩阵内的所有值）

动态规划：

双 dp（土豪购物）——两种情况分析，取或不取，放或不放：

```
a=input()
merch=[int(items) for items in a.split(',') ]
n=len(merch)
dp1=[0]*n
dp2=[0]*n
dp1[0]=merch[0]
dp2[0]=merch[0]
for i in range(1,n):
    dp1[i]=max(dp1[i-1]+merch[i],merch[i]) #不放回
    dp2[i]=max(dp1[i-1],dp2[i-1]+merch[i],merch[i])
print(max(dp2))
```

拦截导弹：

```
 def max_intercepted_missiles(k, heights):
    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)
```

0-1 背包问题（二维 dp）：（N 是物品件数，B 是背包最大承重）

```
N,B=map(int,input().split())
price=[0]+list(map(int,input().split()))
weight=[0]+list(map(int,input().split()))
dp=[[0]*(B+1) for _ in range(N+1)]
for i in range(N+1):
    for j in range(B+1):
        if weight[i]<=j:
            dp[i][j]=max(dp[i-1][j],dp[i-1][j-weight[i]]+price[i])
        else:
            dp[i][j]=dp[i-1][j]
print(dp[-1][-1])
```

最长公共子序列：

```
while True:
    try:
        a, b = input().split()
    except EOFError:
```

```
            break
    alen = len(a)
    blen = len(b)
    dp = [[0]*(blen+1) for i in range(alen+1)]
    for i in range(1, alen+1):
        for j in range(1, blen+1):
            if a[i-1]==b[j-1]:
                dp[i][j] = dp[i-1][j-1] + 1
            else:
                dp[i][j] = max(dp[i-1][j], dp[i][j-1])
    print(dp[alen][blen])
```

多重背包问题（一个物品可以拿多个）：（二进制优化版，虽然不是很懂）

```
def bounded_knapsack_binary(v, w, m, V):
# v: 物品体积列表；  w: 物品价值列表；  m: 每种物品最大数量；  V: 背包容量
    n = len(v)
    items = []
    # 二进制拆分
    for i in range(n):
        count = m[i]
        k = 1
        while k <= count:
            items.append((k * v[i], k * w[i]))
            count -= k
            k *= 2
        if count > 0:
            items.append((count * v[i], count * w[i]))
    # 动态规划
    dp = [0] * (V + 1)
    for volume, value in items:
        for j in range(V, volume - 1, -1):
            dp[j] = max(dp[j], dp[j - volume] + value)
    return dp[V]
# 示例数据
v = [1, 2, 3]  # 物品体积
w = [6, 10, 12]  # 物品价值
m = [3, 2, 1]  # 每种物品最大数量
V = 5  # 背包容量
print(bounded_knapsack_binary(v, w, m, V))  # 输出：30
```

非优化版：当做 0-1 背包做

```
def bounded_knapsack_dp(v, w, m, V):
    n = len(v)
    dp = [0] * (V + 1)
    for i in range(n):
        if m[i] * v[i] >= V:
            # 当物品的总量大于背包容量时，直接按照完全背包处理
            for j in range(v[i], V + 1):
                dp[j] = max(dp[j], dp[j - v[i]] + w[i])
```

```
            else:
                # 否则按照多重背包处理
                for j in range(V, 0, -1):
                    for k in range(1, min(m[i], j // v[i]) + 1):
                        dp[j] = max(dp[j], dp[j - k * v[i]] + k * w[i])
    return dp[V]
```

完全背包问题：

```
n, a, b, c = map(int, input().split())
dp = [-1] * (n + 1)
dp[0] = 0 # 长度为 0 时可以切分 0 段
for i in range(1, n + 1):
    for j in (a,b,c):
        if j<=i and dp[i-j]!=-1:
            dp[i]=max(dp[i],dp[i-j]+1)
print(dp[n])
```

贪心代码：（看一下以右边序列排序的怎么做）

区间问题：

按照右端点排序：重叠区间问题

（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：合并区间问题

（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。

合并区间：给出一个区间的集合，请合并所有重叠的区间。

```
class Solution:
    def merge(self, intervals):
        result = []
        if len(intervals) == 0:
            return result    # 区间集合为空直接返回

        intervals.sort(key=lambda x: x[0])   # 按照区间的左边界进行排序

        result.append(intervals[0])   # 第一个区间可以直接放入结果集中

        for i in range(1, len(intervals)):
            if result[-1][1] >= intervals[i][0]:   # 发现重叠区间
#合并区间只需要更新结果集最后一个区间的右边界，因为根据排序，左边界已经是最小的
                result[-1][1] = max(result[-1][1], intervals[i][1])
            else:
                result.append(intervals[i])   # 区间不重叠
        return result
```

无重叠区间：给定一个区间的集合，找到需要移除区间的最小数量，使剩余区间互不重叠。

```
class Solution:
    def eraseOverlapIntervals(self, intervals: List[List[int]]) -> int:
```

```
        if not intervals:
            return 0
        intervals.sort(key=lambda x: x[0])  # 按照左边界升序排序
        count = 0  # 记录重叠区间数量
        for i in range(1, len(intervals)):
            if intervals[i][0] < intervals[i - 1][1]:  # 存在重叠区间
                intervals[i][1] = min(intervals[i - 1][1], intervals[i][1])  # 更新重叠区间的右边界
                count += 1
        return count
```

充实的寒假生活：

```
n=int(input())
activities=[]
for _ in range(n):
    start,end=map(int,input().split())
    activities.append((start,end))
activities.sort(key=lambda x:x[1])
ed=-1
num=0
for i in range(n):
    if ed<activities[i][0]（即 activities 这个列表里第 i 个元素的 start）:
        num+=1
        ed=activities[i][1]（即 activities 这个列表里第 i 个元素的 end）
print(num)
```

划分字母区间：字符串 S 由小写字母组成。我们要把这个字符串划分为尽可能多的片段，同一字母最多出现在一个片段中。返回一个表示每个字符串片段的长度的列表。

```
s=input()
last_occurrence = {}  # 存储每个字符最后出现的位置
for i, ch in enumerate(s):
    last_occurrence[ch] = i
print(last_occurrence)
result = []
start = 0
end = 0
for i, ch in enumerate(s):
    end = max(end, last_occurrence[ch])  # 找到当前字符出现的最远位置
    if i == end:  # 如果当前位置是最远位置，表示可以分割出一个区间
        result.append(end - start + 1)
        start = i + 1
print(result)
```

应用 Heapq 的示例（毒药）

```
n = int(input())  # 输入药水数量
potion = list(map(int, input().split()))  # 输入药水的效果值
health = 0  # 初始健康值
count = 0  # 能喝的药水数量
min_heap = []  # 小根堆，用来存储已饮用的药水效果
for i in range(n):
    health += potion[i]  # 喝掉当前的药水
```

```
        heapq.heappush(min_heap, potion[i])  # 将药水效果加入堆中
        count += 1  # 增加已喝的药水数量
        # 如果健康值小于 0，说明健康值不合法，需要丢掉一个药水
        if health < 0:
            health -= heapq.heappop(min_heap)  # 弹出最小的药水，回退健康值
            count -= 1  # 减少已喝的药水数量
print(count)
```

Bisect：import 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。

a=bisect.bisect_left(list, item, lo=0, hi=len(list))：在已排序的 list 中查找插入点，使得插入 item 后仍保持列表的有序性。返回的插入点是从列表的左侧开始，第一个大于等于 item 的位置。

例：**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

bisect.bisect_right(list, item, lo=0, hi=len(list)) 或 bisect.bisect()：类似于 bisect_left，但返回的插入点是从列表的右侧开始，第一个大于 item 的位置。

bisect.insort_left(list, item, lo=0, hi=len(list))：将 item 插入到 list 中，使其保持有序性，插入点使用 bisect_left 查找的结果。

例：**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]

bisect.insort_right(list, item, lo=0, hi=len(list)) 或 **bisect.insort()**：将 item 插入到 list 中，插入点使用 bisect_right 查找的结果。

bisect.bisect_left() 和 bisect.bisect_right() 用于查找插入位置。

bisect.insort_left() 和 bisect.insort_right() 用于在保持列表有序的情况下插入元素。

```
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")
```

递归：（找到 base case）

汉诺塔问题：

```
def tower(n,a,b,c):
    if n==1:
        print(a+'->'+c)
        return
    if n>1:
        tower(n-1,a,c,b)
        print(a+'->'+c)
        tower(n-1,b,a,c)
height=int(input())
print(2**height-1)
tower(height,'A','B','C')
```

全排列

（1）使用 Python 内置的 itertools.permutations 方法

```
from itertools import permutations
k = int(input())
l = [i + 1 for i in range(k)]
# 生成全排列
def whole_list(l):
    return permutations(l)
# 打印每种排列
for perm in whole_list(l):
print(" ".join(map(str, perm)))
```

（2）递归的做法：

```
def whole_list(l):
    # 递归终止条件：当列表只有一个元素时，返回该元素的单个排列
    if len(l) == 1:
        return [l]
    # 保存所有排列
    permutations = []
    # 遍历列表中的每个元素
    for i in range(len(l)):
        # 当前选择的元素
        current = l[i]
        # 剩下的元素（去掉当前选中的元素）
        remaining = l[:i] + l[i+1:]
        # 对剩下的元素递归求全排列
        for perm in whole_list(remaining):  （这就是递归调用自身吧）
            # 把当前选中的元素加到每个排列的开头
            permutations.append([current] + perm)
    return permutations
# 输入和调用
k = int(input())
l = [i + 1 for i in range(k)]
# 打印每种排列
for perm in whole_list(l):
    print(" ".join(map(str, perm)))
```

排序：（最大整数类问题 OJ27373：假设有 n 个正整数，要求从中选取几个组成一个位数不超过 m 的新正整数，现需要求出最大可能数值是多少。比如有 4 个数，993，923，3817，3807，组成新数不超过 7 位，那么新数最大值为 9933817。）

```
n = int(input())
num=input.split()
for i in range(n-1):
    for j in range(i+1,n):
        if nums[i]+nums[j]<nums[j]+nums[i]:
            nums[i],nums[j]=nums[j],nums[i]（字典序的比较）
```

```
#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]
```

双指针

（河中跳房子）（对撞指针：回文串）（滑动窗口：最大子数组和、无重复字符的最长子字符串）

```
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
```

回文串：

```
def second(x):
    left = 0
    right = len(x) - 1
    while left <= right:
        if x[left] == x[right]:
            left += 1
            right -= 1
        else:
            return 'No'
    return 'Yes'
```

移除重复的元素（原地修改）：给定一个有序数组，要求原地删除重复的元素，并返回新数组的长度。（其中一个慢指针 slow 用于记录已去重的数组的尾部，另一个快指针 fast 用于遍历整个数组。）

def remove_duplicates(arr):

if not arr:

return 0 如果数组为空，直接返回 0，表示没有元素。

slow = 0 慢指针初始化为 0，表示去重后的新数组的尾部索引。

for fast in range(1, len(arr)): 从索引 1 开始遍历数组，fast 是快指针，用于扫描整个数组。

if arr[fast] != arr[slow]: 如果快指针指向的元素与慢指针指向的元素不同，说明这是一个新的、没有重复的元素。

slow += 1

arr[slow] = arr[fast] 慢指针移动到下一个位置，并将快指针当前的值赋给慢指针所在位置，表示找到一个新的非重复元素。

return slow + 1 慢指针指向的是最后一个非重复元素的索引，所以新数组的长度是 slow + 1。

滑动窗口

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。

例：最长的无重复子序列串

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
```

Stack（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。

应用：快速堆猪

```
stack=[]
min_stack=[]  （一个主栈一个辅助栈）
def push(x):
    stack.append(x)
    if not min_stack or x<= min_stack[-1]:
        min_stack.append(x)
def pop():
    if stack:
        top=stack.pop()
        if top==min_stack[-1]:
            min_stack.pop()
def get_min():
    if min_stack:
        return min_stack[-1]
    return
while True:
    try:
        command=input().strip()
        if command.startswith('push'):
            value=int(command.split()[1])
            push(value)
        elif command.startswith('pop'):
            pop()
        elif command.startswith('min'):
            if get_min() is not None:  （不能直接写 if get_min()因为如果返回值为 0 的话也会视为False）
                print(get_min())
    except EOFError:
        break
```

其它：

与平方数相关的题，平方数可以这么找：

```
squares=set()
i=1
while i*i<=10**9:
    squares.add(i*i)
    i+=1
```

或者列表推导式？squares=[i**2 for i in range(10)]

用 dp 判断某一个字符串是否均由平方数组成：

```
dp=[False]*(n+1)
dp[0]=True
for i in range(1,n+1):
    for j in range(i):
        if int(digits[j:i]) in squares and dp[j]==True:
            dp[i]=True
            break
```

在一定范围内筛选该范围内所有质数的代码：

```
N = 10005
Is_prime = [True] * N
Is_prime[0] = is_prime[1] = False
p = 2
while p * p <= N:
    if is_prime[p]:
        for i in range(p * 2, N, p):
            Is_prime[i] = False
    p += 1
```

找平方素数（恰好有且仅有三个不同的正除数）只需判断该数的平方根是否为素数即可。

前缀和：flower

```
t,k=map(int,input().split())
MAXN=100001
dp=[0]*MAXN
dp[0]=1
sum_flower=[0]*MAXN
MOD = 10**9+7
for i in range(1,MAXN):
    if i<k:
        dp[i]=dp[i-1]
    else:
        dp[i]=(dp[i-1]+dp[i-k])% MOD
    sum_flower[i]=(sum_flower[i-1]+dp[i])% MOD
for _ in range(t):
    a, b = map(int, input().split())
    print((sum_flower[b]-sum_flower[a-1]+MOD)% MOD)
```

求 xx 的最大值之前=都要先设置一个 maxValue = float("-inf")，给最大值的初始值赋为无穷小，这样就可以通过 max(a,b)得出最大值。

将每一个数字转换成单个字符组成的列表 e.g.123→['1', '2', '3']：a = list(str(num))

Kadane's algorithm：Kadane's 算法是一种用于解决最大子数组问题（Maximum Subarray Problem）的高效算法，即在一个一维整数数组中找到具有最大和的连续子数组。

```python
def max_subarray_sum(arr):
    if not arr:
        return 0

    max_current = max_global = arr[0]

    for num in arr[1:]:
        max_current = max(num, max_current + num)
        if max_current > max_global:
            max_global = max_current

    return max_global

# 测试用例
arr = [-2, 1, -3, 4, -1, 2, 1, -5, 4]
print("最大子数组和为:", max_subarray_sum(arr))  # 输出：最大子数组和为: 6
```

两种语言转换类问题：先创建两个字典（分别为字符：数字；数字：字符）；先从第一种语言转换成数字，然后再从数字转换成第二种语言。

```
english=input()
english_to_num_map={'two':2, 'three':3, 'four':4, 'five':5, 'six':6, 'seven':7,'hundred':100, 'thousand':1000, 'million':1000000}
def english_to_num(s):
    total = 0
    current=0
    words = s.split()
    for word in words:
        if word in english_to_num_map:
            value=english_to_num_map[word]
            if value==100:
                current*=value
            elif value>=1000:
                current*=value
                total+=current
                current=0
            else:
                current+=value
    total+=current
    return total
if "negative" in english:
    print("-"+str(english_to_num(english)))
else:
    print(english_to_num(english))
```