二分

```
import bisect

a = [1, 2, 4, 4, 8]
print(bisect.bisect_left(a, 4))  # 输出: 2，获得可以插入的索引

print(bisect.bisect_right(a, 4))  # 输出: 4

print(bisect.bisect(a, 4))  # 输出: 4
Bisect.insort(列表，元素) 往列表中对应位置（按大小顺序）插入指定元素
```

优先队列

即该队列自动的保证顺序

```
import heapq

data = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
heapq.heapify(data)#列表堆化
print(data)  # 输出: [0, 1, 2, 3, 9, 5, 4, 6, 8, 7]

heapq.heappush(data, -5)
print(data)  # 输出: [-5, 0, 2, 3, 1, 5, 4, 6, 8, 7, 9]

print(heapq.heappop(data))  # 输出: -5 弹出堆顶最小元素；只对堆顶进行维护，其他位置是乱的
#想要获得最大堆，可对于所有元素取负
```

如果操作了堆中元素使得其变小，那么可以调用heapq.siftdown保持堆的性质

如果操作了堆中元素使得其变大，那么可以调用heapq.siftup保持堆的性质

```
Heapq.siftdown(堆名称，目标元素的索引
```

日期与时间

```
import calendar, datetime
print(calendar.isleap(2020)) # 输出: True
判断闰年


print(datetime.datetime(2023, 10, 5).weekday())  # 判断星期几输出: 3 (星期四)
```

```
from datetime import date
date1 = date(2024, 12, 26)
date2 = date(2025, 1, 1)
delta = date2 - date1
 print(delta.days) 判断两个日期之间相隔多少天

from datetime import date, timedelta
start_date = date(2024, 12, 26)
days_to_add = 6
end_date = start_date + timedelta(days=days_to_add)
print(end_date) 判断某一天过了若干天之后是哪一天
```

数据结构 双端队列（波兰表达式等）（栈stack）

```
import collections
# deque
dq = collections.deque([1, 2, 3])
dq.append(4)
print(dq)  # 输出: deque([1, 2, 3, 4])
dq.appendleft(0)
print(dq)  # 输出: deque([0, 1, 2, 3, 4])
dq.pop()
print(dq)  # 输出: deque([0, 1, 2, 3])
dq.popleft()
print(dq)  # 输出: deque([1, 2, 3])


dd = collections.defaultdict(int)
（不会index错误的日历，超界的时候默
认返回0）
dd['a'] += 1
print(dd)
 # 输出:defaultdict(<class 'int'>,{'a': 1})
od = collections.OrderedDict()
od['a'] = 1
od['b'] = 2
od['c'] = 3
print(od)  # 输出: OrderedDict([('a', 1), ('b', 2), ('c', 3)])


Point = collections.namedtuple('Point', ['x', 'y'])
p = Point(11, 22)
print(p)  # 输出: Point(x=11, y=22)
print(p.x, p.y)  # 输出: 11 22
```

波兰表达式，不断弹出符号和数字，pop和append操作是核心

```
from typing import List
def calculate(op, num1, num2):
    if op == "+":
        return num1 + num2
    elif op == "-":
        return num1 - num2
    elif op == "*":
        return num1 * num2
    else:
        return num1 / num2
def eval_rpn(tokens: List[str]) -> float:
    stack = []
    for token in reversed(tokens):  # 从右往左
遍历tokens列表
        if token in "+-*/":
            num1 = stack.pop()
            num2 = stack.pop()
            result = calculate(token, num1,
num2)
            stack.append(result)
        else:
            stack.append(float(token))
    return stack[0]
```

小数位数的处理

X1=format(x1_f+x1_b,'.5f')，转换小数位数

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
```

特殊输出：print（*数组）表示输出数组中每一个元素

列表.insert(索引，值)

```
# 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']
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)
#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)
```

```
# 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)
```

想要创建若干个set，应该用l=[set() for x in range(n)],否则会使得所有的集合都是同一个

遍历

```
import itertools

for item in itertools.product('AB', repeat=2):##生成笛卡尔积
    print(item)  # 输出: ('A', 'A'), ('A', 'B'), ('B', 'A'), ('B', 'B')
```

生成排列数 itertools.permutations(列表，长度（不填时默认为全长）)
生成组合数 itertools.combinnations(列表名称，从中选几个)

函数

```
import functools
```

自定义的累积操作，如下为逐步的累加操作

```
print(functools.reduce(lambda x, y: x + y, [1, 2, 3, 4]))  # 输出: 10
```

分数与有理数

```
import fractions, decimal

frac = fractions.Fraction(1, 3)
print(frac)  # 输出: 1/3

dec = decimal.Decimal('0.1')
print(dec)  # 输出: 0.1
```

数学

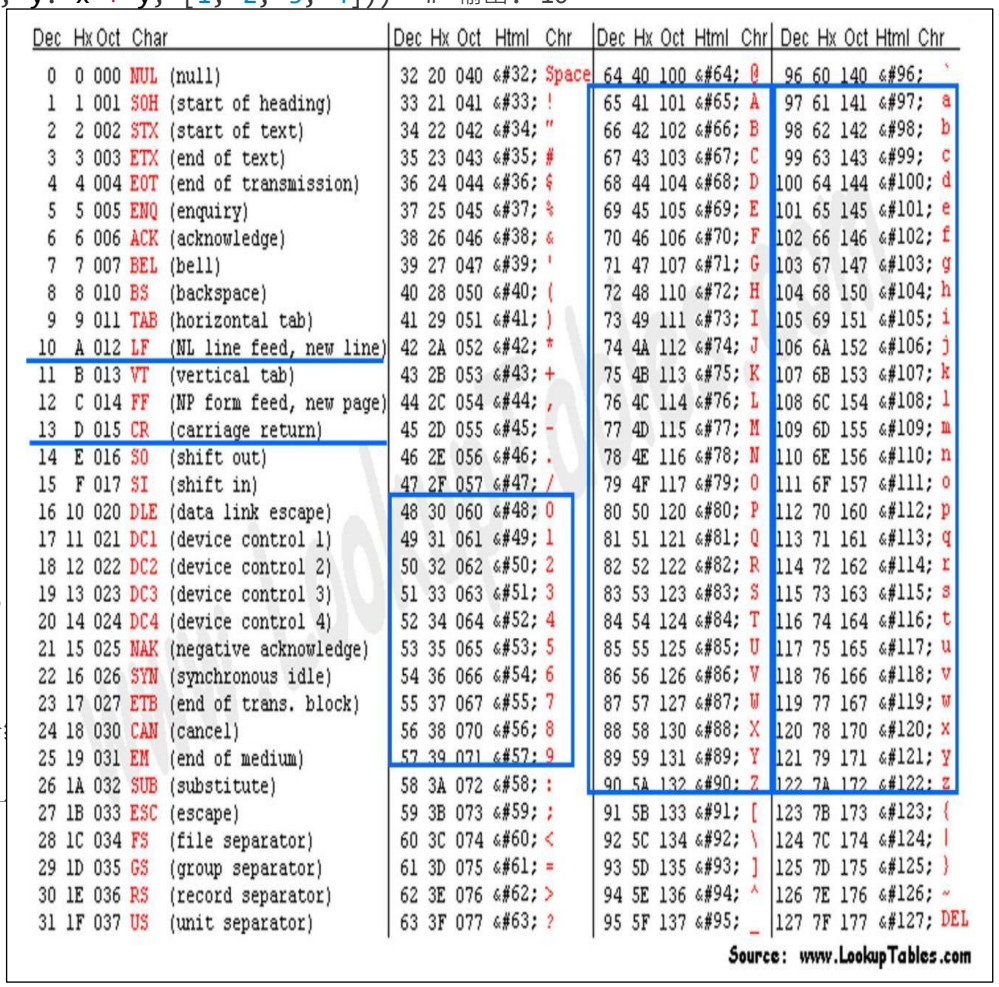
```
import  math
print(math.ceil(4.2))  # 输出: 5
print(math.floor(4.2))  # 输出:
```

Math.gcd(a,b)返回a，b的最大公约数

最小公倍数可以由a*b/（a，b）的最小公
数得到

Math.log(100,10)后一个是底数，若后一
位空着则表示e

Math.π，math.e可以引用常数值



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

拷贝

```
import copy
original = [1, 2, [3, 4]]
copied = copy.deepcopy(original)
print(copied)  # 输出: [1, 2, [3, 4]]
```

字符串操作函数

.upper()##全大写 .lower()全小写 .capitalize()字符串中多个单词，但只有第一个单词首字母大写，title()多个单词，每个都首字母大写

ASCII码

Ord()读取码

Chr()按照码输出

Print('\n')换行符，print(****,end=' ')涉及输出结尾不换行的情况

数组操作

```
squared = list(map(lambda x: x**2, [1, 2, 3, 4]))
print(squared)  # 输出: [1, 4, 9, 16]

a = [1, 2, 3]
b = ['a', 'b', 'c']
zipped = list(zip(a, b))合并两个列表生成由元组组成的列表
print(zipped)  # 输出: [(1, 'a'), (2, 'b'), (3, 'c')]

filtered = list(filter(lambda x: x > 2, [1, 2, 3, 4]))
print(filtered)  # 输出: [3, 4]##过滤筛，满足才入列

enumerated = list(enumerate(['a', 'b', 'c'])) ,
print(enumerated)  # 输出: [(0, 'a'), (1, 'b'), (2, 'c')]
```

产生索引值和列表对应值的元组对

算法

dfs模板

连通域染色

```
directions = ((0, 1), (0, -1), (-1, 0), (1, 0),
              (1, -1), (1, 1), (-1, -1), (-1, 1))


def dfs(x, y):
    global n, m, color, board
    board[x][y] = color
    area = 1
    for d in directions:
        nx = x+d[0]
        ny = y+d[1]
        if any((nx < 0, ny < 0, nx >= n, ny >= m)):
            continue
        if board[nx][ny] == "W":
            area += dfs(nx, ny)
    return area
```

水淹七军

```
def dfs_iterative(x, y, h, m, n, grid, water_height):
    stack = [(x, y)]
    water_height[x][y] = h
```

```
        while stack:
            cx, cy = stack.pop()


            for dx, dy in directions:
                nx = cx + dx
                ny = cy + dy
                if (0 <= nx < m and 0 <= ny < n and
                    grid[nx][ny] < h and
                    water_height[nx][ny] < h):
                    water_height[nx][ny] = h
```

bfs模板

螃蟹采蘑菇

```
from collections import deque
def bfs(x1,y1,x2,y2,graph,m,n):
    dx = [-1, 1, 0, 0]
    dy = [0, 0, -1, 1]
    queue = deque([((x1,y1),(x2,y2), 0)])
    # (left, right, step)
    visited = [[False] * n for _ in range(m)]
    visited[x1][y1] = True
    while queue:
        left, right, step = queue.popleft()

        if graph[left[0]][left[1]] == 9\
            or graph[right[0]][right[1]]== 9:
            return 'yes'

        for i in range(4):
            new_left=(left[0]+dx[i],left[1]+dy[i])
            new_right=(right[0]+dx[i],right[1]+dy[i])

            if 0 <= new_left[0] < m and 0 <= new_left[1] < n\
                and 0 <= new_right[0] < m and 0 <= new_right[1] < n\
                and not visited[new_left[0]][new_left[1]]\
                 and graph[new_left[0]][new_left[1]]!= 1\
                and graph[new_right[0]][new_right[1]]!=1:
                # and not visited[new_right[0]][new_right[1]]\

                visited[new_left[0]][new_left[1]] = True
                # visited[new_right[0]][new_right[1]] = True
                queue.append((new_left, new_right, step + 1))

    return 'no’
```

欧拉筛法求素数(可选两个不同的方式生成不同列表)

```
def euler_sieve(n):
    is_prime = [True] * (n + 1)
    primes = []
    for i in range(2, n + 1):
        if is_prime[i]:
            primes.append(i)
        for p in primes:
            if i * p > n:
                break
            is_prime[i * p] = False
            if i % p == 0:
                break
    return primes


n = 50
print(euler_sieve(n))  # 输出: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

```
def eulerS(n):
  is_prime=[True]*(n+1)
  is_prime[0]=is_prime[1]=False
  prime=[]
  for i in range(2,n+1):
      if is_prime[i]:
          prime.append(i)
      for pr in prime:
          if i*pr>n:
              break
          is_prime[pr*i]=False
          if i%pr==0:
              break
  return is_prime  ## is_prime[2]=true,is_prime[3]=True
```

dp

01背包

```
def knapsack(weights, values, capacity):
    n = len(weights)#capacity是装载能力
    dp = [[0] * (capacity + 1) for _ in range(n + 1)]


    for i in range(1, n + 1):
        for w in range(1, capacity + 1):
            if weights[i - 1] <= w:
                dp[i][w] = max(dp[i - 1][w], dp[i - 1][w - weights[i - 1]] + values[i - 1])
            else:
                dp[i][w] = dp[i - 1][w]
    return dp[n][capacity]
weights = [1, 2, 3, 4]
values = [10, 20, 30, 40]
capacity = 5
print(knapsack(weights, values, capacity))  # 输出: 50
```

完全背包

```
def knapsack_complete(weights, values, capacity):
    dp = [0] * (capacity + 1)
    for i in range(len(weights)):
        for w in range(weights[i], capacity + 1):
            dp[w] = max(dp[w], dp[w - weights[i]] + values[i])
    return dp[capacity]

weights = [1, 2, 3, 4]
values = [10, 20, 30, 40]
capacity = 5
print(knapsack_complete(weights, values, capacity))  # 输出: 50
```

必须装满的完全背包

```
def knapsack_complete_fill(weights, values, capacity):
    dp = [-float('inf')] * (capacity + 1)
    dp[0] = 0
    for i in range(len(weights)):
        for w in range(weights[i], capacity + 1):
            dp[w] = max(dp[w], dp[w - weights[i]] + values[i])
    return dp[capacity] if dp[capacity] != -float('inf') else 0

weights = [1, 2, 3, 4]
values = [10, 20, 30, 40]
capacity = 5
print(knapsack_complete_fill(weights, values, capacity))  # 输出: 50
```

多重背包（二进制优化）

```
def binary_optimized_multi_knapsack(weights, values, quantities, capacity):
    n = len(weights)
    items = []
    #  将每个物品拆分成若干子物品
    for i in range(n):
        w, v, q = weights[i], values[i], quantities[i]
        k = 1
        while k < q:
            items.append((k * w, k * v))
            q -= k
            k <<= 1
        if q > 0:
            items.append((q * w, q * v))
    # 动态规划求解01背包问题
    dp = [0] * (capacity + 1)
    for w, v in items:
        for j in range(capacity, w - 1, -1):
```

```
                dp[j] = max(dp[j], dp[j - w] + v)
        return dp[capacity]

weights = [1, 2, 3]
values = [6, 10, 12]
quantities = [10, 5, 3]
capacity = 15

print(binary_optimized_multi_knapsack(weights, values, quantities, capacity))  # 输出: 120
```

最长公共子序列问题

▶ 参考代码

```
def lcs(str1, str2, dp):
    len1 = len(str1)
    len2 = len(str2)

    for i in range(1, len1+1):
        for j in range(1, len2+1):
            #判断对应字符，字符串索引从0开始，dp表从1开始
            if str1[i-1] == str2[j-1]:
                dp[i][j] = dp[i-1][j-1] + 1
            else:
                dp[i][j] = max(dp[i-1][j], dp[i][j-1])

    return dp[len1][len2]

str1 = "bdcaba"
str2 = "abcbda"

len1 = len(str1)
len2 = len(str2)

dp = [[0 for i in range(len1+1)] for j in range(len2+1)]

a = lcs(str1, str2, dp)

print("最长公共子序列长度为", a)
```

```
def getlcs(str1, str2, dp):
    len1 = len(str1)
    len2 = len(str2)

    i = len1
    j = len2

    res = ""

    while i != 0 and j != 0:
        if str1[i-1] == str2[j-1]: #字符相同 记录 移向左上角
            res += str1[i-1]
            i -= 1
            j -= 1
        else:                      #字符不同 判断从左还是上边来
            if dp[i][j] == dp[i-1][j]:
                i -= 1
            else:
                j -= 1

    return res[::-1]

b = getlcs(str1, str2, dp)

print("最长公共子序列为", b)


def printf(dp, str1, str2):
```

由已经得到的dp表反求最长公共子串的内容

公共子序列

Dijkstra

```
import heapq

def dijkstra(graph, start):
    n = len(graph)
    distances = {node: float('inf') for node in range(n)}
    distances[start] = 0
    priority_queue = [(0, start)]

    while priority_queue:
        current_distance, current_node = heapq.heappop(priority_queue)
        if current_distance > distances[current_node]:
            continue
        for neighbor, weight in graph[current_node]:
            distance = current_distance + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(priority_queue, (distance, neighbor))
    return distances
graph = {
    0: [(1, 4), (2, 1)],
    1: [(3, 1)],
    2: [(1, 2), (3, 5)],
    3: []
}

start_node = 0
print(dijkstra(graph, start_node))  # 输出: {0: 0, 1: 3, 2: 1, 3: 4}
```

岛屿最短距离dijkstra，由一个一堆1的岛屿到另一个一堆1的岛屿

```
import heapq##堆结构很重要
def dijkstra(x1,y1):
    q,visited=[],[[False]*m for _ in range(n)]
    heapq.heappush(q,(0,x1,y1))
    while q:
        step,x,y=heapq.heappop(q)，（0，x，y）0需要在最前面
        if visited[x][y]:
            continue
        visited[x][y] = True
        if ma[x][y]==1 and step!=0:
            return step
        for dx,dy in dire:
            if 0<=x+dx<n and 0<=y+dy<m and not visited[x+dx][y+dy]:
                heapq.heappush(q,(step+1-ma[x+dx][y+dy],x+dx,y+dy))
```

走山路（高度差为距离）（所经历的高度差最小即为距离最小）

```
import heapq
dire = [(0, 1), (0, -1), (1, 0), (-1, 0)]
def dijkstra(x1, y1, x2, y2):##参数为起点坐标和终点坐标
    q = []
    visited = [[False] * n for _ in range(m)]
    heapq.heappush(q, (0, x1, y1))
    while q:
        step, x, y = heapq.heappop(q)
        if visited[x][y]:
            continue
```

```
        visited[x][y] = True
        if x == x2 and y == y2:
            return step
        for dx, dy in dire:
            nx, ny = x + dx, y + dy
            if 0 <= nx < m and 0 <= ny < n and not visited[nx][ny] and ma[nx][ny] !=
'#':
                new_step = step + abs(int(ma[x][y]) - int(ma[nx][ny])) if ma[x][y] !=
'#' else step
                heapq.heappush(q, (new_step, nx, ny))
    return ‘NO’
```

```
m, n = map(int, input().split())
    dp = [[0] * (n + 1) for _ in
range(m + 1)]
    for i in range(m + 1):
        for j in range(1, n +
1):
            if i == 0 or j == 1:
                dp[i][j] = 1
            elif i < j:
                dp[i][j] =
dp[i][i]
            else:
                dp[i][j] =
dp[i][j - 1] + dp[i - j][j]
    print(dp[m][n])
```

放苹果，dpij即为i个苹果放入j个盘子

旋律问题

```
n,m=map(int,input().split())
tones=list(map(int,input().split()))
cnt=0##count
s=set()
for i in tones:
    s.add(i)
    if len(s)==m:
        s.clear()
        cnt+=1
print(cnt+1)
```

##每次出现一到五的一组，即表示更多一位的选择可以被满足
##如13452；24153；的两组可以满足所有的两位音符组成，那么遍历完成之后，就能得到结果
极其精妙的集合使用，考虑给定种类和数位的种类问题

滑雪dfs和dp

```
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)
```

修建建筑，不交叉情况下最大叠放（区间）

```
def generate_intervals(x, width, m):
    temp = []
    for start in range(max(0, x-width+1),
min(m, x+1)):
        end = start+width
        if end <= m:
            temp.append((start, end))
    return temp
intervals.sort(key=lambda x: (x[1], x[0]))
cnt = 0
last_end = 0
for start, end in intervals:
    if start >= last_end:
        last_end = end
        cnt += 1
```

割绳子，insort插入的用法

```
from bisect import insort
n=int(input())
lines=list(map(int,input().split()))
lines.sort()
while len(lines)!=1:
    a=lines[0]+lines[1]
    ans+=a
    del lines[0]del lines[0]
    insort(lines,a)
print(ans)
```

Basketball exercise交叉轮换型dp

```
n=int(input())
h1=list(map(int,input().split()))
h2=list(map(int,input().split()))
dp1=[0]*(n)
dp2=[0]*(n)
dp1[0]=h1[0]
dp2[0]=h2[0]
for i in range(1,n):
    dp1[i]=max(dp2[i-1]+h1[i],dp1[i-1])
    dp2[i]=max(dp1[i-1]+h2[i],dp2[i-1])
print(max(dp1[-1],dp2[-1]))
```

##世界杯只因 区间完全覆盖所需最小数值

```
def calculate_min_coverage(n, points):
    clips = [(max(0, i-point), min(n-1,
i+point))
             for i, point in
enumerate(points)]
    clips.sort()
    st, ed = 0, n-1
    res = 0
    current_index = 0
    while current_index < n:
        maxR = -float("inf")
        while current_index < n and
clips[current_index][0] <= st:
            maxR = max(maxR,
clips[current_index][1])
            current_index += 1
        if maxR < st:
            break
        res += 1
        if maxR >= ed:
            break
        st = maxR + 1##不断更新
    return res
```

上山观景题目的两侧走的dp

```
dp1=[1]*(n)
dp2=[1]*(n)
##假设一直上山
for i in range(n):
    for j in range(i):
        if hs[i]>hs[j]:
            dp1[i]=max(dp1[i],dp1[j]+1)
        else:
            continue
#从右往左假设一直下山
for i in range(n-1,-1,-1):
    for j in range(n-1,i,-1):
        if hs[i]>hs[j]:
            dp2[i]=max(dp2[i],dp2[j]+1)
        else:
            continue
ans=0
for i in range(n):
    ans=max(ans,dp1[i]+dp2[i]-1)
```

```
def is_valid_parentheses(s)
    stack = []
    # 括号映射，括号匹配问题
    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
```

反悔型遍历，堆的应用

```
import heapq
def try_drink(n,potions):
    hp=0
    used=[]
    for potion in potions:
        hp+=potion
        heapq.heappush(used,potion)
        if hp<0:
            hp-=used[0]
            heapq.heappop(used)
    return len(used)
n=int(input())
potions=map(int,input().split())
ans=try_drink(n, potions)
print(ans)
```

最佳凑单的dp

```
n, t = map(int, input().split())
p = list(map(int, input().split()))

# 计算所有商品价格总和，用于确定dp数组第二维的大小上限
total_price = sum(p)
Dp[i][j]表示用i个商品凑j钱 能否凑出
# 创建dp数组并初始化边界条件
dp = [[False] * (total_price + 1) for _ in range(n + 1)]
dp[0][0] = True

for i in range(1, n + 1):
    for j in range(total_price + 1):
        dp[i][j] = dp[i - 1][j] or (j >= p[i - 1] and dp[i - 1][j - p[i - 1]])

# 从目标凑单价格t开始查找能凑出且最接近t的价格
for j in range(t, total_price + 1):
    if dp[n][j]:
        print(j)
        break
else:
    print(0)
```

Kadane算法

最大子序列：通过一次遍历数组来高效地找到最大子序列和，它基于这样一个观察：对于每个位置 i 上的元素，以它结尾的最大子序列和要么是当前元素本身（即前面的子序列和为负数，抛弃前面的，重新从当前元素开始），要么是包含前面元素的子序列和再加上当前元素（即前面的子序列和为正数，继续累加当前元素）。以下代码可以查找和最大的连续子序列

```python
def max_subarray_sum(nums):
    max_sum = current_sum = nums[0]
    for num in nums[1:]:
        current_sum = max(num, current_sum + num)
        max_sum = max(max_sum, current_sum)
    return max_sum


nums = [-2, 1, -3, 4, -1, 2, 1, -5, 4]
print(max_subarray_sum(nums))  # 输出: 6
```

最大子矩阵

枚举 l~r 列的子式，按行求和后，求这些和的最大子序列即可。

```python
def kadane_1d(arr):
    """
    一维Kadane算法，用于求给定一维数组中的最大子序列和，一维的这一段和上面的kadane是一样的
    """
    max_end_here = max_so_far = arr[0]
    for num in arr[1:]:
        max_end_here = max(num, max_end_here + num)
        max_so_far = max(max_so_far, max_end_here)
    return max_so_far


def max_submatrix_sum(matrix):
    """
    求二维矩阵中的最大子矩阵和
    """
    if not matrix:
        return 0
    rows = len(matrix)
    cols = len(matrix[0])
    max_sum = float('-inf')
    for l in range(cols):
        for r in range(l, cols):
            row_sums = [0] * rows
            for row in range(rows):
                for col in range(l, r + 1):
                    row_sums[row] += matrix[row][col]
            current_sum = kadane_1d(row_sums)
            max_sum = max(max_sum, current_sum)
    return max_sum
```

最长上升子序列

dp方法

```python
def length_of_lis(nums):
    if not nums:
        return 0
    dp = [1] * len(nums)
    for i in range(1, len(nums)):
        for j in range(i):
            if nums[i] > nums[j]:
                dp[i] = max(dp[i], dp[j] + 1)
    return max(dp)


nums = [10, 9, 2, 5, 3, 7, 101, 18]
print(length_of_lis(nums))  # 输出: 4
```

类似的需要求最长下降子序列的长度的题目；如跳高，准备切割木头这样的存在明显的各待处理元素间比较关

系影响输出结果的题目。
Dilworth定理:
Dilworth定理表明，任何一个有限偏序集的最长反链(即最长下降子序列)的长度，
等于将该偏序集划分为尽量少的链(即上升子序列)的最小数量。
因此，计算序列的最长下降子序列长度，即可得出最少需要多少台测试仪。
"""

```
from bisect import bisect_left
def min_testers_needed(scores):
    scores.reverse()  # 反转序列以找到最长下降子序列的长度，即为跳高题目中需要的测试仪数目
    lis = []  # 用于存储最长上升子序列
    for score in scores:
        pos = bisect_left(lis, score)
        if pos < len(lis):
            lis[pos] = score
        else:
            lis.append(score)
    return len(lis)
```

切割木材问题dilworth(可以先排序)

```
#答案就是对l排序后求w的最长严格递减子序列（用Dilworth's theorem不难证明）#两个参数的类dilworth
# 最长严格递减子序列有经典的nlogn的算法
#一般有一样的都不是大问题，因为可以把(3,5) (3,6) 直接看作(3.1, 5) (3.2, 6)
import bisect
def doit():
    n = int(input())
    data = list(map(int, input().split()))
    sticks = [(data[i], data[i + 1]) for i in range(0, 2 * n, 2)]##木头长度，重量
    sticks.sort()
    f = [sticks[i][1] for i in range(n)]
    f.reverse()
    stk = []
    for i in range(n):
        t = bisect.bisect_left(stk, f[i])
        if t == len(stk):
            stk.append(f[i])
        else:
            stk[t] = f[i]
    print(len(stk))
T = int(input())
for _ in range(T):
    doit()
```

二分法

```
import bisect


def length_of_lis_binary(nums):
    if not nums:
        return 0
    tails = []
    for num in nums:
        pos = bisect.bisect_left(tails, num)
        if pos == len(tails):
            tails.append(num)
        else:
            tails[pos] = num
    return len(tails)


nums = [10, 9, 2, 5, 3, 7, 101, 18]
print(length_of_lis_binary(nums))  # 输出: 4
```

下一个全排列

```
def next_permutation(nums):
i = len(nums) - 2
while i >= 0 and nums[i] >= nums[i + 1]:
  i -= 1
if i >= 0:
  j = len(nums) - 1
  while nums[j] <= nums[i]:
     j -= 1
  nums[i], nums[j] = nums[j], nums[i]
nums[i + 1:] = reversed(nums[i + 1:])
return nums
   nums = [1, 2, 3]
   print(next_permutation(nums))  # 输出: [1, 3, 2]
```

最长公共子串问题

▶ 参考代码

```
def lcSubstr(str1, str2, dp):
    len1 = len(str1)
    len2 = len(str2)

    maxlen = 0

    for i in range(1, len1+1):
        for j in range(1, len2+1):
            #判断对应字符，字符串索引从0开始，dp表从1开始
            if str1[i-1] == str2[j-1]:
                dp[i][j] = dp[i-1][j-1] + 1
                maxlen = max(dp[i][j], maxlen)
            else:
                dp[i][j] = 0

    return maxlen

str1 = "bdcaba"
str2 = "abcbda"

len1 = len(str1)
len2 = len(str2)

dp = [[0 for i in range(len2+1)] for j in range(len1+1)]

a = lcSubstr(str1, str2, dp)
```

迭代计算最大值

不等就重新计数

```
def lcSubstr(str1, str2, dp):
    len1 = len(str1)
    len2 = len(str2)

    maxlen = 0
    maxj = 0

    for i in range(1, len1+1):
        for j in range(1, len2+1):
            #判断对应字符，字符串索引从0开始，dp表从1开始
            if str1[i-1] == str2[j-1]:
                dp[i][j] = dp[i-1][j-1] + 1
                if maxlen < dp[i][j]:
                    maxlen = dp[i][j]
                    maxj = j
            else:
                dp[i][j] = 0

    #字符串索引从0开始，dp表从1开始，所以dp里的j用在字符串里需要减1
    #所以本来应该是从maxj-maxlen+1到maxj，减1后变为maxj-maxlen到maxj-1
    #又python中分片右侧取不到，所以右侧需要加1，所以为maxj-maxlen到maxj
    substr = str2[maxj-maxlen:maxj]
    print("最长公共子串为", substr)

    return maxlen
```

由已经得到的dp表反求最长公共子串

Manacher算法 处理最长回文子串

```python
def manacher(s):
    s = '#' + '#'.join(s) + '#'
    n = len(s)
    p = [0] * n
    c = r = 0
    for i in range(n):
        mirr = 2 * c - i
        if i < r:
            p[i] = min(r - i, p[mirr])
        while i + p[i] + 1 < n and i - p[i] - 1 >= 0 and s[i + p[i] + 1] == s[i - p[i] - 1]:
            p[i] += 1
        if i + p[i] > r:
            c, r = i, i + p[i]
    max_len, center_index = max((n, i) for i, n in enumerate(p))
    return s[center_index - max_len:center_index + max_len].replace('#', '')

s = "babad"
print(manacher(s))  # 输出: "bab" 或 "aba"
```

最大钢条切割利润

```python
def cut_rod(p, n):
    if n == 0:
        return 0
    q = 0
    for i in range(1, n+1):
        q = max(q, p[i]+cut_rod(p, n-i))

    return q

p = [0, 1, 5, 8, 9, 10, 17, 17, 20, 24, 30]

print(cut_rod(p, 10))
```

矩阵连乘

```python
def matrixmulchain(m, i, j):
    if i == j:
        return 0
    c = matrixmulchain(m, i, i) + matrixmulchain(m, i+1, j) + m[i][0]*m[i][1]*m[j][1]
    for k in range(i+1, j):
        tmp = matrixmulchain(m, i, k) + matrixmulchain(m, k+1, j) + m[i][0]*m[k][1]*m[j][1]
        if tmp < c:
            c = tmp
    return c


m = [[5, 10], [10, 50], [50, 2], [2, 5]]
res = matrixmulchain(m,0,3)

print(res)
```

装箱问题

```
import math
restforb = [0,5,3,1]
while True:
    a,b,c,d,e,f = map(int,input().split())
    if a + b + c + d + e + f == 0:
        break
    boxes = d + e + f
    boxes += math.ceil(c/4)
    space_for_b = 5*d + restforb[c%4]
    if b > space_for_b:
          boxes += math.ceil((b - space_for_b)/9)
    space_for_a = boxes*36 - (36*f + 25*e + 16*d + 9*c + 4*b)

    if a > space_for_a:
        boxes += math.ceil((a - space_for_a)/36)
    print(boxes)
```

约瑟夫问题一：由人数n和倍数k来得出幸存编号

```
while True:
    n, m = map(int, input().split())
    if n + m == 0:
        break
    a = 1
    for i in range(2, n + 1):
        a = (a + m - 1) % i + 1
    print(a)
```

进程检测，用最少的检查点去检查所有区间

```
k = int(input())
for _ in range(k):
    n = int(input())
    tasks = []
    for _ in range(n):
        s, d = map(int,
input().split())

        tasks.append((s, d))
    sorted_tasks = sorted(tasks,
key=lambda x: (x[1],-x[1]+x[0]))
    count = 1
    current_time = sorted_tasks[0][1]
    for s, d in sorted_tasks:
        if s > current_time:
            count += 1
            current_time = d
    print(count)
```

约瑟夫问题二：逐步输出出圈的人

```
while True:
    n,p,m = map(int, input().split())
    check=[True]*(n+1)
    child=n
    if n+p+m==0:
        break
    count=0
    for i in range(p,36000):
        if i%n!=0:
            if check[i%n]==True and child > 1:
                count+=1
                if count==m:
                    check[i%n]=False
                    print(str(i%n)+',',end='')
                    child-=1
                    count=0

            elif check[i%n]==False:
                continue
            elif check[i%n] and child==1:
                check[i%n]=False
                child=0
                print(str(i%n))
                break
        elif i%n==0:
            if check[n]==True and child>1:
                count+=1
                if count==m :
                    check[n]=False
                    print(str(n)+',',end='')
                    count=0
                    child-=1
            elif check[n]==False:
                continue
            elif check[n] and child ==1:
                check[n]=False
                child=0
                print(str(n))
                break
```

二维矩阵卷积运算，m,n;p,q分别为两个矩阵的长宽

```
m,n,p,q=map(int,input().split())
big=[list(map(int,input().split())) for x in range(0,m)]
small=[list(map(int,input().split())) for y in range(0,p)]
results=[[0]*(n-q+1) for z in range(m-p+1)]
for i in range(m-p+1):
    for j in range(n-q+1):
        for s in range(p):
            for t in range(q):
                results[i][j]+=small[s][t]*big[s+i][t+j]
for k in range(m-p+1):
    print(' '.join(map(str,results[k])))
```

```
while i < a and j < b:###双指针在集合加法中的使用
        sum_val = A[i] + B[j]
        if sum_val > s:
            i += 1
        elif sum_val < s:
            j += 1
        else:
            # 统计相同元素的数量
            a_count = 1
            b_count = 1
            while i + a_count < a and A[i + a_count] == A[i]:
                a_count += 1
            while j + b_count < b and B[j + b_count] == B[j]:
                b_count += 1
            count += a_count * b_count
            i += a_count
            j += b_count
    print(count)
```

处理整数摆放排列问题的排序（st:str）

```
def biggerS(st):
    return st*10
```

导弹拦截，求所有

```
k = int(input())
heights = list(map(int,
input().split()))
# 每个位置的导弹为结尾的最长递减序列长度
dp = [1] * k
for i in range(k):
    for j in range(i):
        if heights[i] <=
heights[j]:
            dp[i] = max(dp[i],
dp[j] + 1)

print(max(dp))
```

```
#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]
```

```
weight=[]#每个元素的位数
for num in l:
    weight.append(len(num))
#dp[i][j]在前i数中选择，不超过j位，最大可能数值
dp=[['']*(m+1) for _ in range(n+1)]
for k in range(m+1):
    dp[0][k]=''#无法组成整数
for q in range(n+1):
    dp[q][0]=''#无法组成整数
for i in range(1,n+1):
    for j in range(1,m+1):
        if weight[i-1]>j:#不能选第i个，因为会超位数
            dp[i][j]=dp[i-1][j]
        else:#可以选第i个也可以不选
                dp[i][j]=str(max(f(dp[i-
1][j]),int(l[i-1]+dp[i-1][j-weight[i-1]])))
print(dp[n][m])
```

十进制转换成k进制

通过不断地用十进制数除以目标进制数 k，取余数和商，直到商为 0，然后将余数从下往上排列即可得到 k 进制数。可以使用内置函数divmod()来同时获取除法的商和余数，方便计算。

导弹拦截问题及最长递减链的获取

```
k = int(input())
heights = list(map(int, input().split()))
# 每个位置的导弹为结尾的最长递减序列长度
dp = [1] * k
for i in range(k):
    for j in range(i):
        if heights[i] <= heights[j]:
            dp[i] = max(dp[i], dp[j] + 1)

print(max(dp))
```

剪彩带，和凑零钱同理，目的是将总长度分得尽可能多

```
n, a, b, c = map(int, input().split())
proble = [a, b, c]
proble.sort()
def find(length):
    dp = [float('-inf')] * (length + 1)
    dp[0] = 0
    for size in proble:
        for i in range(size, length + 1):
            dp[i] = max(dp[i], dp[i - size] + 1)
    return dp[length]
print(find(n))
```

八皇后问题生成皇后列表

```
def available(board, row, col):
    for i in range(row):
        if board[i] == col or abs(board[i] - col) ==
abs(i - row):
            return False
    return True
def solve_eight_queens():
    def back(row):
        if row == 8:
            solutions.append(list(board))
            return
        for col in range(8):
            if available(board, row, col):
                board[row] = col
                back(row + 1)
    board = [-1] * 8
    solutions = []
    back(0)
    return solutions
n = int(input())
solutions = solve_eight_queens()
for _ in range(n):
    b = int(input())
    result = [str(solutions[b - 1][i] + 1) for i in
range(8)]
    print(''.join(result))
```

给定一组n种不同面额的硬币，以及要支付的总金额

计算并返回可以凑成总金额所需的 最少的硬币个数 。如果没有任何一种硬币组合能组成总金额，返回 -1。

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

coins=list(map(int,input().split()))

coins.sort(reverse=True)

dp=[float('inf')]*(total+1)

dp[0]=0

for coin in coins:

    for i in range(coin,total+1):

        dp[i]=min(dp[i],dp[i-coin]+1)

if dp[total]!=float('inf'):

    print(dp[total])

else:

    print(-1)
```

a矩阵*b矩阵得到c矩阵的运算代码

```
for i in range(n):
    for j in range(n):
        for k in range(n):
            c[i][j]+=a[i][k]*b[k][j]
```

摆动序列的dp，双dp表维护

```
dp=[[1,1] for x in range(n)]
for i in range(1,n):
    if nums[i]>nums[i-1]:
        dp[i][1]=dp[i-1][0]+1
        dp[i][0]=dp[i-1][0]
    elif nums[i]<nums[i-1]:
        dp[i][0]=dp[i-1][1]+1
        dp[i][1]=dp[i-1][1]
    else:
        dp[i][0]=dp[i-1][0]
        dp[i][1]=dp[i-1][1]
ans=max(dp[n-1][0],dp[n-1][1])
```

奶牛跳石头的最大最小值问题的二分查找模板

```
def check(x):
    num = 0
    tip = 0
    for i in range(1, n+2):
        if rock[i] - tip < x:
            num += 1
        else:
            tip = rock[i]
    if num > m:
        return True
    else:
        return False
start, end = 0, L+1
ans = -1
while start < end:
    mid = (start + end) // 2
    if check(mid)==True:
        end = mid
    else:          # 返回False，有可能是num==m
        ans = mid # 如果num==m, mid可能是答案
        start = mid + 1
```

月度开销预算的最大最小值（与左边类似的二分）

```
def check(budgt):
    count = 0
    fajo = 1
    for i in range(n):
        count += costs[i]
        if count > budgt:
            fajo += 1
            count = costs[i]
        if fajo > m or (i == n - 1 and
count > budgt):  # 考虑最后一个月开销也不能
超预算
            return True
    return False
while start <= end:
    mid = (start + end) // 2
    if check(mid):
        start = mid + 1
    else:
        end = mid - 1
print(start)
```

回溯类型dp

```
n, k = map(int, input().split())
    values=[0]
    locas=[0]
    locas += list(map(int,
input().split()))
    values += list(map(int,
input().split()))
    dp = [0] * (n + 1)  # 这里dp长度设为n +
1，方便对应每个地点的状态
    dp[0]=0
    dp[1] = values[1]  # 初始化第一个地点的
最大利润为其本身利润
    for i in range(1, n+1):
        # 记录上一个符合距离要求的地点索引，
初始化为0
        prev_index = 0
        for j in range(i):
            if locas[i] - locas[j] > k:
                prev_index = j
            else:
                break
        dp[i] = max(dp[prev_index] +
values[i], dp[i - 1])
    print(max(dp))#这种涉及相隔距离的dp需要
在每个数据点回溯一段来做判断
```

将一个字符串经删、换、加三种操作变为另一个的最少次数

```
def edit_distance(s1, s2):
    m, n = len(s1), len(s2)
    # 创建dp数组并初始化边界条件
    dp = [[0] * (n + 1) for _ in range(m +
1)]
    for i in range(m + 1):
        dp[i][0] = i
    for j in range(n + 1):
        dp[0][j] = j

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if s1[i - 1] == s2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1]
            else:
                dp[i][j] = min(dp[i - 1][j -
1], dp[i][j - 1], dp[i - 1][j]) + 1
#min里面的三个比较分别对应替换，删除，添加
###如果只可以删除操作，则dp[i][j] = min(
dp[i][j - 1], dp[i - 1][j]) + 1
    return dp[m][n]
s1,s2=map(str,input().split())
print(edit_distance(s1, s2))
```

螺旋序列的转动实现

```
n=int(input())
side=[[401]*(n+2)]
mx=side+[[401]+[0]*n+[401] for x in
range(n)]+side
move=[[0,1],[1,0],[0,-1],[-1,0]]
f_x=1
f_y=1
x,y=move[0]
count=0
for i in range(1,n**2+1):
    mx[f_x][f_y]=i
    if mx[f_x+x][f_y+y]:
        count+=1
        x,y=move[count%4]
    f_x+=x
    f_y+=y
for j in range(1,n+1):
    print(*mx[j][1:n+1])
```

吃flowersdp模型，预求和处理

```
for i in range(1,10**5+1):
    if i>=k:
        dp[i]=(dp[i-1]+dp[i-
k])%(10**9+7)
    else:
        dp[i]=1
    pre_sum[i]=pre_sum[i-1]+dp[i]
for _ in range(t):
    a,b=map(int,input().split())
    print((pre_sum[b]-pre_sum[a-
1]+10**9+7)%(10**9+7))
    ##pre_sum中是对dp中数取模后再求和
，因而不是严格递增，
    ##可能出现负数取模，故而要加上
10**9+7
```

count=0马走日dfs求步数走法的模板

```
def dfs(x, y, n, m, step, field):
    global count##全局变量的使用
    if step == n * m:
        count += 1
        return
    dx = [2, 2, -2, -2, 1, 1, -1, -1]
    dy = [1, -1, 1, -1, 2, -2, 2, -2]
    for i in range(8):
        new_x = x + dx[i]
        new_y = y + dy[i]
        if 0 <= new_x < n and 0 <= new_y <
m and not field[new_x][new_y]:
            field[new_x][new_y] = True
            dfs(new_x, new_y, n, m, step +
1, field)
            field[new_x][new_y] = False
            ##关键
t = int(input())
for _ in range(t):
    n, m, x, y = map(int, input().split())
    field = [[False] * m for _ in range(n)]
    field[x][y] = True
    dfs(x, y, n, m, 1, field)
     print(count)  count=0#初始化
```

基于deque的bfs迷宫模板（三维visited）

```
from collections import deque
def bfs(m, n, graph, k, start_x, start_y):
    dx = [-1, 1, 0, 0]
    dy = [0, 0, -1, 1]
    queue = deque([(start_x, start_y, 0)])  # (x,
y, time)
    visited = [[[False] * k for _ in range(n)] for
_ in range(m)]
    # 三维数组记录每个位置在每个时间点是否被访问过
    while queue:
        x, y, time = queue.popleft()
        if graph[x][y] == 'E':
            return time
        for i in range(4):
            new_x = x + dx[i]
            new_y = y + dy[i]
            if 0 <= new_x < m and 0 <= new_y < n:
                new_time = (time + 1) % k
                if graph[new_x][new_y] != '#' or
new_time == 0:
                    if not
visited[new_x][new_y][new_time]:

visited[new_x][new_y][new_time] = True
                        queue.append((new_x,
new_y, time + 1))

    return 'Oop!'
```

土豪游戏，dp1可以是为最大连续字串和，dp2是可以放回一个的情况

```
dp1=[0]*(l)
dp2=[0]*(l)
dp1[0]=vs[0]
dp2[0]=vs[0]
for i in range(1,l):
    dp1[i]=max(dp1[i-1]+vs[i],vs[i])
    dp2[i]=max(dp1[i-1],dp2[i-1]+vs[i],vs[i])
print(max(dp2))
```

n块鸡排，k个锅的greedy模型

```
n,k=map(int,input().split())
times=list(map(int,input().split()))
total=sum(times)
maxtime=total/k
times.sort()
if times[-1]>maxtime:
    for i in range(n-1,-1,-1):
        if times[i]<=maxtime:
            break
        total-=times[i]
        k-=1
        maxtime=total/k
print(f"{maxtime:.3f}")
```