

## 字符串

`str.title()`首字母大写（每个单词） `str.lower()/upper()`每个字母小/大写 `str.strip()`去除空格，有`rstrip/lstrip`去掉尾部/头部的空格

`ord()` `chr()` 可以完成字符与ASCII码的转化

`str.find()`查找指定字符，注意如果说有的话会返回第一个找到的，如果没有会返回-1

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

Source: [www.LookupTables.com](http://www.LookupTables.com)

字典：`defaultdict` 是 Python 中 `collections` 模块提供的一个字典（`dict`）的子类，功能上与普通字典类似，但它具有一个默认值工厂函数，可以在访问不存在的键时自动为该键创建一个默认值，而不会抛出 `KeyError`。

如果访问一个不存在的键，`defaultdict` 会自动使用你提供的工厂函数来创建一个默认值，而普通字典在这种情况下会抛出 `KeyError`。你可以指定工厂函数，例如 `list`、`int`、`set` 等，或者自定义一个工厂函数来返回任何默认值。

```
from collections import defaultdict
```

常见的工厂函数包括：

- `int`: 默认值为 0
- `list`: 默认值为空列表 []
- `set`: 默认值为空集合 `set()`
- `str`: 默认值为空字符串 ""
- `float`: 默认值为 0.0

字典对象的方法	含    义
<code>dict.keys()</code>	返回包含字典所有 <code>key</code> 的列表
<code>dict.values()</code>	返回包含字典所有 <code>value</code> 的列表
<code>dict.items()</code>	返回包含所有(键,值)项的列表
<code>dict.clear()</code>	删除字典中的所有项或元素，无返回值
<code>dict.copy()</code>	返回字典浅复制副本
<code>dict.get(key,default=None)</code>	返回字典中 <code>key</code> 对应的值，若 <code>key</code> 不存，则返回 <code>default</code> 的值 ( <code>default</code> 默认为 <code>None</code> )
<code>dict.pop(key[,default])</code>	若字典中存在 <code>key</code> ，则删除并返回 <code>key</code> 对应的 <code>value</code> ；如果 <code>key</code> 不存在，且没有给出 <code>default</code> 值，则引发 <code>KeyError</code> 异常
<code>dict.setdefault(key,default=None)</code>	若字典中不存在 <code>key</code> ，则由 <code>dict[key]=default</code> 为其赋值
<code>dict.update(aDict)</code>	将字典 <code>aDict</code> 中键值对添加到 <code>dict</code> 中

集合对象的方法	含    义
<code>set.add(x)</code>	向集合中添加元素 <code>x</code>
<code>set.update(a_set)</code>	使用集合 <code>a_set</code> 更新原集合
<code>set.pop()</code>	删除并返回集合中的任意元素
<code>set.remove(x)</code>	删除集合中的元素 <code>x</code> ，如果 <code>x</code> 不存在则报错
<code>set.discard(x)</code>	删除集合中的元素 <code>x</code> ，如果 <code>x</code> 不存在则什么也不做
<code>set.clear()</code>	清空集合中的所有元素

操作	含    义
<code>&lt;seq&gt;[i]</code>	索引（求 <code>&lt;seq&gt;</code> 中位置索引为 <code>i</code> 的元素）
<code>&lt;seq&gt;[i:j:k]</code>	切片（求 <code>&lt;seq&gt;</code> 的位置索引为 <code>i~j-1</code> 的子列表）
<code>&lt;seq1&gt;+&lt;seq2&gt;</code>	将 <code>&lt;seq1&gt;</code> 和 <code>&lt;seq2&gt;</code> 连接
<code>&lt;seq&gt;*&lt;int-expr&gt;或&lt;int-expr&gt;*&lt;seq&gt;</code>	将 <code>&lt;seq&gt;</code> 复制 <code>&lt;int-expr&gt;</code> 次
<code>len(&lt;seq&gt;)</code>	求 <code>&lt;seq&gt;</code> 长度
<code>for&lt;var&gt;in&lt;seq&gt;:</code>	对 <code>&lt;seq&gt;</code> 中元素循环
<code>&lt;expr&gt;in&lt;seq&gt;</code>	查找 <code>&lt;seq&gt;</code> 中是否存在 <code>&lt;expr&gt;</code> ，返回值为布尔类型
<code>del &lt;seq&gt;</code>	删除列表
<code>del &lt;seq&gt;[i]</code>	删除列表中位置索引为 <code>i</code> 的元素
<code>max(&lt;seq&gt;)</code>	返回列表中最大值
<code>min(&lt;seq&gt;)</code>	返回列表中最小值

列表，字典

append以及pop（都是O(1)的）。但是注意del, remove, pop(0), insert, index等都是O(n)的！反复remove很有可能导致超时，一般开一个真值表先打标记

切片操作关于所切长度是线性复杂度，反复切片很可能超时；切片list[k:l]当k>=l时不会报错而是返回空列表

in list也是线性复杂度！尽量避免in list的判断，必要时最好用dict或set代替。

list.index()慎用，不仅是线性复杂度，而且在找不到的时候会抛出IndexError

Python特性：允许负数下标，正数越界才会报错。

注意函数有无返回值：list.sort(),list.reverse()都是原地修改而不返回，如要用返回值需用sorted()和reversed()（注意reversed()返回的不是列表而是reversed对象，如需用列表要用list转换类型，但是for循环则不需要转换）

排序list.sort(key=lambda x:x[0],reverse=True) True :3 2 1

list.sort(key=lambda x: (x[0],x[1])) 现根据x[0]排序，再根据x[1]排序

sum([])会返回0，但是min/max([])会报错！

列表解析式：[f(x) for x in list (if P(x))] 创建列表

列表可以相加list.extend()

遍历列表的时候通常用for循环；但是尽量避免一边循环一边删除/添加元素。如果必须的话建议改用while循环。

zip函数

```
a = [1, 2, 3]
b = ['a', 'b', 'c']
zipped = list(zip(a, b))
print(zipped) # 输出: [(1, 'a'), (2, 'b'), (3, 'c')]
```

浅拷贝问题：

基本原则：大于1维很可能出问题，尽量不要用\*或者copy拷贝一个列表，不要在不同的地方引用同一个列表变量

例如[[0] \* m]不会有问题，但是[[[0] \* m] \* n]会出问题，可以写成[[0] \* m for \_ in range(n)]

拷贝一维列表可以a = b[:]复制一份，因为a=b的话a和b会指向同一个列表

二维使用copy模块中的deepcopy深拷贝

## 字典

key和value都可以是任意类型的东西（不过key一定要是不可变的，比如list不能作为key；但用list作为value是非常常见的）。

把有些信息用字典提前存下来，在需要的时候直接访问，有时是降低时间复杂度的好办法。但是有MLE的风险

如果要按顺序遍历通常用for x in sorted(dict.keys()/values())

dict.keys()/values()/items()分别返回键/值/键值对列表

dict.get(key,default=None)查找指定key的value, 如果key不存在不会报错而是返回给定的默认值

dict.pop(key,default=None)弹出指定键值对

如果key已经存在, 再赋值时会直接覆盖。

## 枚举enumerate

for i,x in enumerate(list),遍历list中的 (下标, 值)

对math库语法: 【1】对数和指数相关的函数: math.log(x, base): 计算以指定底base的对数, 默认底为e;

- math.log2(x): 计算以 2 为底的对数
- 【2】幂和平方根: •math.pow(x, y): 计算  $x^y$  ( $x$  的  $y$  次幂)
- math.sqrt(x): 返回  $x$  的平方根
- 【3】•math.factorial(x): 返回  $x!$  ( $x$  的阶乘)
- math.gcd(x, y): 计算  $x$  和  $y$  的最大公约数
- 【4】•math.ceil(x) 和 math.floor(x): •math.ceil(x): 返回大于等于  $x$  的最小整数 (向上取整)
- math.floor(x): 返回小于等于  $x$  的最大整数 (向下取整)

itertools.product([0, 1], repeat=6) 来生成所有可能的 6 位长度的二进制组合。 (元组)

**itertools** 常用功能 【1】 排列与组合: itertools 提供了生成序列排列和组合的工具函数:

- permutations(iterable, r)•用于生成可迭代对象的排列
- 返回长度为  $r$  的排列, 如果未指定  $r$ , 默认为  $\text{len}(iterable)$
- 每个排列都是一个元组。

```
from itertools import permutations
data = [1, 2, 3]
print(list(permutations(data))) # 全排列
print(list(permutations(data, 2))) # 长度为2的排列</p>
[(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)]
[(1, 2), (1, 3), (2, 1), (2, 3), (3, 1), (3, 2)]
```

## 迭代器库itertools

```
import itertools
for item in itertools.product('AB', repeat=2): # 生成笛卡尔积
    print(item) # ('A', 'A')\n('A', 'B')\n('B', 'A')\n('B', 'B')
for item in itertools.product('AB', '12'):
    print(item) # ('A', '1')\n('A', '2')\n('B', '1')\n('B', '2')
result = list(permutations([1,2,3])) # [(1, 2, 3), (1, 3, 2), .....];生成全排列
```

```

from itertools import permutations
def check(word):
    for perm in permutations(dicts, len(word)):
        for i, d in enumerate(perm):
            if word[i] not in d:
                break
        else:
            return 'YES'
    else:
        return 'NO'

```

- combinations(iterable, r)
- 用于生成可迭代对象的组合 • 组合不考虑顺序，且每个元素只会被选一次
- combinations\_with\_replacement(iterable, r)
- 与 combinations 类似，但允许同一个元素被选多次

## 【2】笛卡尔积

- product(\*iterables, repeat=1) • 用于生成输入可迭代对象的笛卡尔积。

```

from itertools import product
data1 = [1, 2]
data2 = ['A', 'B']# 之间与自身的笛卡尔积
print(list(product(data1, data2)))#[(1, 'A'), (1, 'B'), (2, 'A'), (2, 'B')]
print(list(product(data1, repeat=2))) #[(1, 1), (1, 2), (2, 1), (2, 2)]

```

```

from functools import cmp_to_key
m = int(input()) # 最大位数
n = int(input()) # 数字的数量
x = list(input().split()) # 输入的数字列表
def cmp(a, b):# 自定义比较函数，用于排序数字
    if a + b > b + a:
        return -1 # 如果a+b大于b+a, a应该排在前面
    elif a + b < b + a:
        return 1 # 如果a+b小于b+a, b应该排在前面
    else:
        return 0 # 如果相等，顺序无关
# 排序数字列表，使得拼接后的结果最大
x.sort(key=cmp_to_key(cmp))

# 一维动态规划数组，dp[i]表示组成i位的最大整数
dp = [''] * (m + 1)
# 动态规划填充dp数组
for num in x:
    length = len(num)
    # 从后往前更新dp数组，防止重复计算
    for j in range(m, length - 1, -1):

```

```

# 如果可以通过拼接当前num更新dp[j], 则更新
if dp[j - length] != '' or j == length:#⚠⚠⚠只有dp[j - length] != '' or j == length的时候才更新列表, 使得明确去除不可达的情况, 只对可达的情况进行更新⚠⚠⚠
    dp[j] = max(dp[j], dp[j - length] + num)

if dp[m] == '':# 输出结果: 如果dp[m]为空, 返回更小位数中最大的一位
    # 查找dp中非空的最大值, 且位数最接近m
    for k in range(m - 1, 0, -1):
        if dp[k] != '':
            print(dp[k])
            break
    else:
        print(dp[m])

```

**string操作** string.replace(old, new); string.split(); string.strip() 移除首位指定字符,默认为空格/换行; string.find(item) 会输出第一个位置,或给出 -1.

**list/tuple操作** list('abc')==['a','b','c']; list(range(4))==[0,1,2,3]  
len(list); list.append(item); list.extend(list\_Extend); list.insert(index, item); list.clear(self); list.reverse(self)  
list.pop(index),返回 index 处元素的值 list.index(item) 会输出第一个位置,或抛出 ValueError .可以通过 try...except 结构捕捉.  
list.sort(key = None, reverse = False) 原地更改,返回 None .只可用于list list\_sorted = sorted(list) 返回排序后的列表,也可用于tuple/string "str\_join".join(list\_str) (将序列中元素以指定字符串连接) zip() 压缩可迭代对象

```

a = [1, 2]
b = ['A', 'B', 'C']
c = [True, False, True]
zipped = list(zip(a, b, c)) # [(1, 'A', True), (2, 'B', False)]
e, f, g = zip(*zipped) # (1, 2), ('A', 'B'), (True, False)

```

**filter(function, iterable)** 过滤可迭代对象

```

numbers = [1, 2, 3, 4, 5, 6]
result = list(filter(lambda n: n % 2 == 0, numbers)) # [2, 4, 6]

```

list[::-1] 得到反转后的 list list[a:b+1] 返回从索引值为 a 到 b-1 的列表型结果,无法应用原地算法,正确做法e.g.:

```

list[a:b+1] = reversed(list[a:b+1])

```

1. 浅拷贝/深拷贝 声明二维数组时, .copy() 导致所有行引用同一个列表,从而修改一个元素时牵连影响.建议列表推导式或深拷贝.

```

matrix[0][0] = 1
print(matrix) # 输出: [[1, 0, 0, 0], [1, 0, 0, 0], [1, 0, 0, 0]]
m = 3; n = 4 # 列表推导式
matrix = [[0 for _ in range(n)] for _ in range(m)]
print(matrix) # 输出: [[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]

```

```

from copy import deepcopy
lcopy=deepcopy(l)

```

```
3. deque from collections import deque, .popleft(), .pop, .appendleft(), .append()
```

4. namedtuple 定义字段名称, 不止可以用索引访问

```
from collections import namedtuple
Point = namedtuple('Point', ['x', 'y'])
p = Point(10, 20)
print(p.x) # 10
print(p.y) # 20
print(p) # Point(x=10, y=20)
```



**set操作** s=set() 创建无序不重复空集合/集合化list/tuple/dict. s.add(s); x (not) in s; s.remove(x) 删除, 可能抛出 KeyError; s.discard(x) 删除, 如不存在不抛出异常; s.clear(); s.pop() 移除随机元素, 会返回元素值, 对空集合抛出 KeyError

1. 集合运算: a|b 并; a&b 交; a-b 差( $x \in a, x \notin b$ ); a^b 对称差( $=a|b-a\&b$ ); <( $=$ ), >( $=$ ) 表包含关系

**dict操作** key只能0维的, 不可是复合数据类型. del d[key]; d.pop(key) 返回 value, 可能抛出 KeyError; d.popitem() 随机弹出键值对元组; d.clear() 清空; d.get(key, dft=None) 可能返回 dft 但不报错; d.keys(); d.values() >> dict\_keys([items]).

1. defaultdict: 会为缺失的键提供一个默认值(int->0, list->[], set->set(), str->""), 而不是抛出 KeyError

```
from collections import defaultdict
dictionary = defaultdict(list) # 自定义默认值defaultdict(lambda: XXX)
```



如果访问不存在的键, 则输出异常. 会带有类型标记, 输出时需要转换格式.

(index, item)生成器enumerate enumerate(list, tuple, string), 输出一些tuple. 欲对item排列, 需转化为新列表:

```
indexed_list1 = list(enumerate(list1))
indexed_list1.sort(key=lambda x: x[1], reverse = True) # False升序, True降序; 也可直接-x[1]
```

二分插入内置函数bisect import bisect, 重要的两个方法如下: bisect\_left(a, x, lo=0, hi=len(a)) 返回x在升序列表a中的插入位置(左起寻找第一个满足条件的位置, right相反)

```
print(bisect.bisect_left([1, 3, 4, 7, 9], 5)) # >>>3
```

insort\_left(a, x, lo=0, hi=len(a)) 将x从左侧插入a, 并保证a的有序性

随机数:

```
import random
x=random.randint(a,b)#>=a,<=b的随机整数
x=random.random()#0~1随机浮点数
x=random.uniform(a,b)#a,b之间随机浮点数
x=random.choice(list)#在列表list中随机选择
```

深拷贝:

```
from copy import deepcopy
lcopy=deepcopy(l)
```

## 保留小数

```
"{:.2f}".format(num) #或  
"%..2f" % num
```

建一个以每行输入的字符串为元素的列表:`sys.stdin.readlines()`

遍历每一项的索引和值组成的元组(类似于`dict.items()`):`for index,value in enumerate(list)`

无穷大:`float('inf')`

`lru_cache`(用来存储每次递归的结果,遇到相同自变量的递归时直接返回这个结果,因此它不能在此时再进行更新其中的全局变量的操作, `dfs`常用):

```
from functools import lru_cache  
@lru_cache(maxsize = 128)
```

循环正常结束后执行的语句:`while...else...`

递归深度限制设置:`sys.setrecursionlimit(...)`

位运算: `a<<b`就是在`bin(a)`(a的二进制表示)后加b个0

```
import math #矩阵  
def find_max_one(matrix,n):  
    max_one=0  
    x=math.ceil(n/2)  
    for layer in range(x):  
        current_result=0  
        top,bottom,left,right=layer,n-layer-1,layer,n-layer-1  
        if left==right:  
            current_result+=matrix[top][left]  
            max_one=max(max_one,current_result)  
            break  
        for i in range(top,bottom+1):  
            current_result+=matrix[i][left]  
            current_result+=matrix[i][right]  
            for j in range(left+1,right): #eg.调用 range(4, 3) 不会报错, 而是返回一个空的 range 对象。  
                current_result+=matrix[left][j]  
                current_result+=matrix[right][j]  
            max_one=max(max_one,current_result)  
    print(max_one)
```

```

n=int(input())
board=[list(map(int,input().split())) for _ in range(n)]
find_max_one(board,n)

```

## 二、dp

dp中“正难则反”的常见类型 【1】 区间问题:从目标区间反向分解为子区间 【2】 路径问题:从终点反向推导到起点, 利用已有结果 【3】 子序列问题:从末尾回溯, 逐步构建解

```

T, M = map(int, input().split())#0-1背包问题(Knapsack problem)。
dp = [ [0] + [0]*T for _ in range(M+1) ]
t = [0]; v = [0]
for i in range(M):
    ti, vi = map(int, input().split())
    t.append(ti)
    v.append(vi)
for i in range(1, M+1):      # 外层循环 (行) 药草M
    for j in range(0, T+1): # 内层循环 (列) 时间T
        if j >= t[i]:
            dp[i][j] = max(dp[i-1][j], dp[i-1][j-t[i]] + v[i])
        else:
            dp[i][j] = dp[i-1][j]
print(dp[M][T])

```

空间优化的一维数组解法

```

T,M = map(int,input().split())
herb = []
for i in range(M):
    herb.append([int(x) for x in input().split()])
dp=[0]*(T+1)
for i in range(M):
    for j in range(T,0,-1):
        if j >= herb[i][0]:
            dp[j] = max(dp[j],dp[j-herb[i][0]]+herb[i][1])
print(dp[-1])

```

## 多重背包

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”

```

dp = [0]*T
for i in range(n):
    all_num = nums[i]
    k = 1
    while all_num>0:
        use_num = min(k,all_num) #处理最后剩不足2的幂的情形
        for t in range(T,use_num*time[i]-1,-1):
            dp[t] = max(dp[t-use_num*time[i]]+use_num*value[i],dp[t])
        k *= 2
        all_num -= use_num

```

注：背包问题的DP解法需要时间T不太大，因为要遍历每个可能的T。如果T很大而物品数量很少，采用DFS枚举物品的选法有时是更好的选择

区间dp:dp[i][j] 表示区间 [i, j] 的最优解 状态转移:拆分区间dp[i][j]=min/max{dp[i][k]+dp[k+1][j]+cost(i,j,k)}(i<=k<j),从小到大递推. 预处理:利用前缀和等快速计算区间cost;结合特殊性质优化(否则复杂度为O(n^3))

```

n = int(input()) # 石子的堆数
stones = list(map(int, input().split()))
sum_ = [0] * (n + 1) # 前缀和, 用于快速计算区间和
for i in range(1, n + 1):
    sum_[i] = sum_[i - 1] + stones[i - 1]
dp = [[float('inf')] * n for _ in range(n)] # dp 数组, 初始化为正无穷
for i in range(n):
    dp[i][i] = 0 # 单堆的代价为 0
for L in range(2, n + 1): # 枚举区间长度 L, 从 2 到 n
    for i in range(n - L + 1): # 起始位置从0到n-L
        j = i + L - 1 # 长度为L后的终点
        for k in range(i, j): # 枚举分割点 k
            dp[i][j] = min(dp[i][j], dp[i][k] + dp[k + 1][j] + sum_[j + 1] - sum_[i])
print(dp[0][n - 1])

```

```

#摆动序列
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))

```

## 最长公共子序列

```
for i in range(len(A)):
    for j in range(len(B)):
        if A[i] == B[j]:
            dp[i][j] = dp[i-1][j-1]+1
        else:
            dp[i][j] = max(dp[i-1][j],dp[i][j-1])
```

## 最长单调子序列

```
dp = [1]*n
for i in range(1,n):
    for j in range(i):
        if A[j]<A[i]:
            dp[i] = max(dp[i],dp[j]+1)
ans = sum(dp)
```

```
#土豪购物，双dp，分别考虑with和no
def max_value(arr):
    n=len(arr)
    if n==1:
        return arr[0]
    else:
        dp_no_remove=[0]*n
        dp_with_remove=[0]*n

        dp_no_remove[0]=arr[0]
        dp_with_remove[0]=-float('inf')
        max_sum=arr[0]

        for i in range(1,n):
            dp_no_remove[i]=max(arr[i],dp_no_remove[i-1]+arr[i])
            dp_with_remove[i]=max(dp_no_remove[i-1],dp_with_remove[i-1]+arr[i])
            max_sum=max(max_sum,dp_no_remove[i],dp_with_remove[i])
        return max_sum
a=list(map(int,input().split(',')))
print(max_value(a))
```

```
def count_nuclear_material_placements_dp(N, M):
    """
    计算在长度为 N 的路径中放置核电站的合法方式数，要求连续放置核电站的最大数量不能超过 M。参数：N - 路径的总长度（格子数）。M - 连续放置核电站的最大限制。返回值：合法放置核电站的方式总数。
    """
    # dp[i][j]: 表示长度为 i 的路径中，最后一个位置连续放置 j 个核电站的合法方案数
    dp = [[0] * M for _ in range(N + 1)]
```

```

dp[0][0] = 1 # 初始状态：长度为 0 的路径只有一种放置方式，即不放置核电站。

# 遍历路径长度，从 1 到 N
for i in range(1, N + 1):
    # 计算当前路径长度下，最后一个位置不放核电站的方案数
    # 这相当于前一个位置的所有状态 (dp[i-1][k], k 从 0 到 M-1) 的总和
    dp[i][0] = sum(dp[i-1][k] for k in range(M))

    # 计算当前路径长度下，最后一个位置连续放 j 个核电站的方案数
    for k in range(1, M):
        # 如果最后一个位置连续放 k 个核电站，
        # 那么 i-1 的最后一个位置必须连续放 k-1 个核电站
        dp[i][k] = dp[i-1][k-1]

# 最终结果是长度为 N 的路径中，所有合法方案数的总和
# 包括最后一个位置不放核电站 (dp[N][0]) 或连续放置 1 到 M-1 个核电站 (dp[N][1], ..., dp[N][M-1])
return sum(dp[N][k] for k in range(M))

a, b = map(int, input().split())
print(count_nuclear_material_placements_dp(a, b))# 输出总的合法方案数

```

两分法，分成不同情况

```

s=input();n=len(s)
dp=[[0 for i in range(2)] for j in range(n)]

#####dp[i][0/1]表示前i个考虑完之后，并且前i个都是R/B需要的最少次数
if s[0]=='R':
    dp[0][0],dp[0][1]=0,1
else:
    dp[0][0],dp[1][0]=1,0
for i in range(1, n):
    if s[i]=='R':
        dp[i][0],dp[i][1]=min(dp[i-1][0],dp[i-1][1]+2), min(dp[i-1][0],dp[i-1][1])+1
    else:
        dp[i][0],dp[i][1]=min(dp[i-1][0],dp[i-1][1])+1, min(dp[i-1][0]+2,dp[i-1][1])
print(dp[n-1][0])

```

三、贪心

```

n,m=map(int,input().split())
location=[list(map(int,input().split()))for _ in range(n)]
possible=[]
count=0
current=0
for i in range(n):
    must,width=location[i][0],location[i][1]
    for start in range(max(must-width+1,0),min(must,m-width)+1):

```

```

possible.append((start,start+width))
possible.sort(key=lambda x:(x[1],x[0]))
for left,right in possible:
    if left>=current:
        count+=1
        current=right
print(count)

```

```

def calculate_min_coverage(n, intervals):
    # 计算每个起点的最远结束点。
    ends = [0]*n
    for i in range(n):
        left_index = max(i - intervals[i], 0)
        ends[left_index] = max(ends[left_index], i + intervals[i] + 1)
    # 初始化左边和右边的边界指针以及区间计数器。
    left_boundary, right_boundary = -1, 0
    cover_count = 0
    # 用贪心算法寻找最少的区间覆盖完全部点。
    #while right_boundary < n:
        # 获取当前最大覆盖范围。
        # new_right_boundary = max(ends[left_boundary + 1: right_boundary + 1])
        # 优化：跟踪当前的最大值，而不是对列表进行切片
        current_max = 0
        while right_boundary < n:# 只在必要时扫描ends数组，确定新的右边界
            if right_boundary >= current_max:
                for i in range(left_boundary + 1, right_boundary + 1):
                    current_max = max(current_max, ends[i])
            new_right_boundary = current_max
            # 更新左边界和右边界。
            left_boundary, right_boundary = right_boundary, new_right_boundary
            # 更新覆盖区间计数。
            cover_count += 1
    return cover_count
n = int(input())
intervals = list(map(int, input().split()))
print(calculate_min_coverage(n, intervals))

```

```

#后悔贪心,greedy,heapq
#greedy:77ms,16KB
import heapq
n=int(input())
lst=list(map(int,input().split()))
health=0
count=0
min_heap=[]
for potion in lst:
    health+=potion
    count+=1
    if potion<0:

```

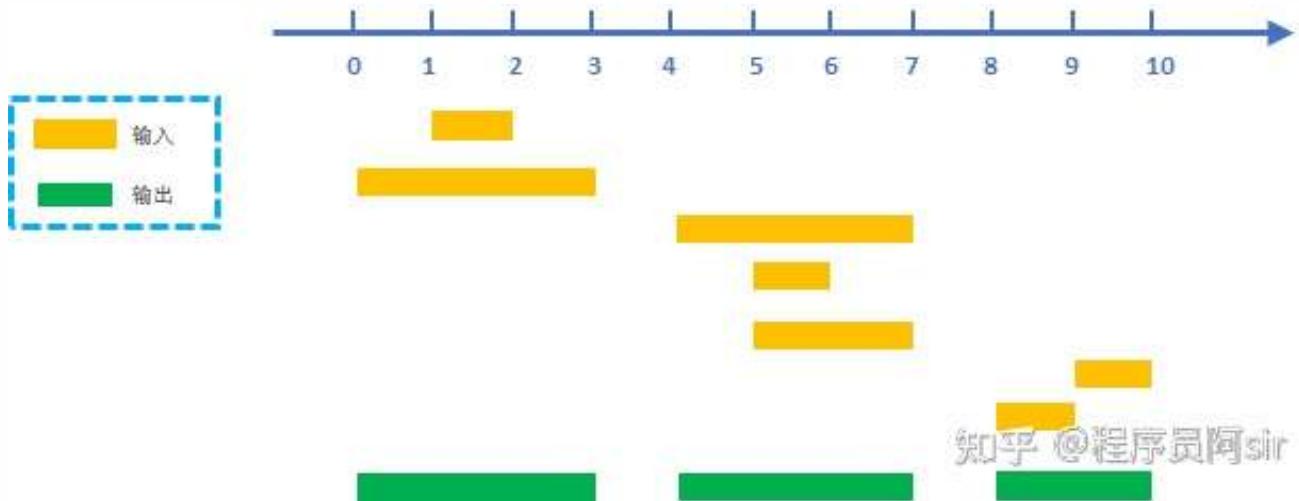
```

    heapq.heappush(min_heap, potion)
if health<0:
    count-=1
    health-=heapq.heappop(min_heap)
print(count)

```

## 1 区间合并

给出一堆区间，要求合并所有有交集的区间（端点处相交也算有交集）。最后输出合并之后的区间。



知乎 @程序员阿sir

**【步骤一】**：按照区间左端点从小到大排序 **【步骤二】**：维护前面区间中最右边的端点为ed。从前往后枚举每个区间，判断是否应该将当前区间视为新区间；假设当前遍历到的区间为第i个区间  $[l_i, r_i]$ ，有以下两种情况：1】  $l_i < ed$ : 说明当前区间与前面区间有交集。因此不需要增加区间个数，但需要设置  $ed = \max(ed, r_i)$ ; 2】  $l_i > ed$ : 说明当前区间与前面没有交集。因此需要增加区间个数，并设置  $ed = \max(ed, r_i)$

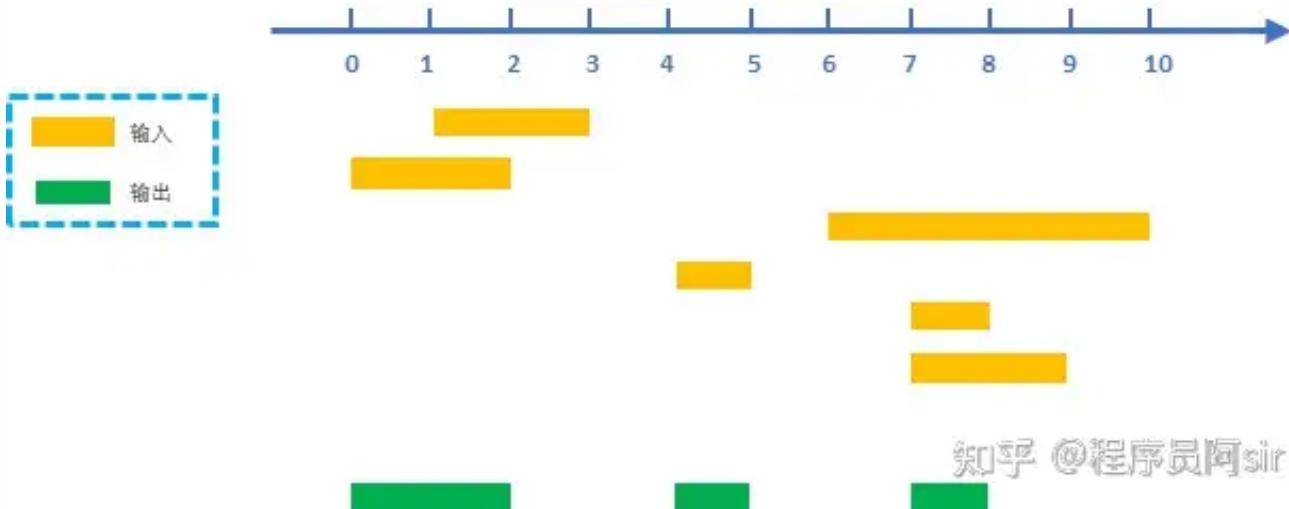
```

list.sort(key=lambda x:x[0])
st=list[0][0]
ed=list[0][1]
ans=[]
for i in range(1,n):
    if list[i][0]<=ed:
        ed=max(ed,list[i][1])
    else:
        ans.append((st,ed))
        st=list[i][0]
        ed=list[i][1]
ans.append((st,ed))

```

## 2 选择不相交区间

要求选择尽量多的区间，使得这些区间互不相交，求可选取的区间的最大数量。这里端点相同也算有重复

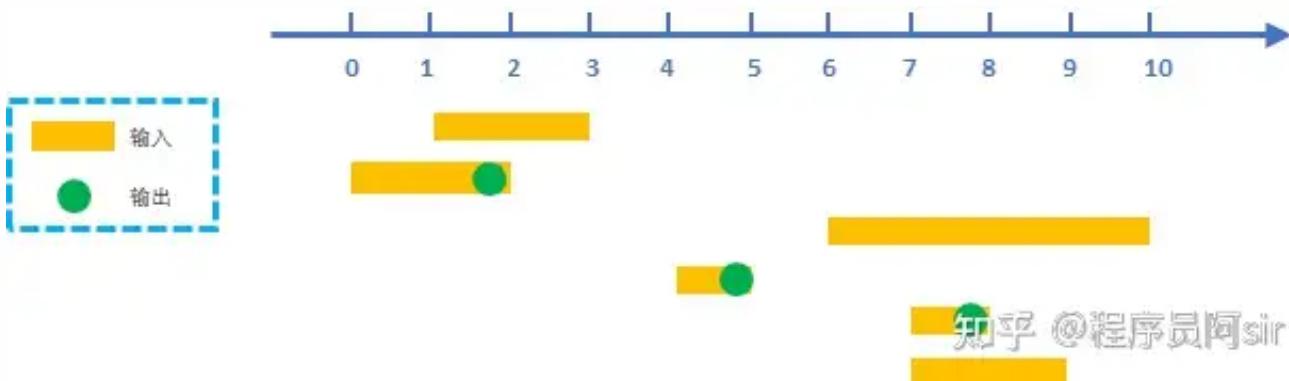


【步骤一】：按照区间右端点从小到大排序。【步骤二】：从前往后依次枚举每个区间。假设当前遍历到的区间为第*i*个区间  $[l_i, r_i]$ ，有以下两种情况：1】 $l_i < ed$ : 说明当前区间与前面区间有交集。因此直接跳过。2】 $l_i > ed$ : 说明当前区间与前面没有交集。因此选中当前区间，并设置  $ed = r_i$ 。

```
list.sort(key=lambda x:x[1])
ed=list[0][1]
ans=[]
for i in range(1,n):
    if list[i][0]<=ed:
        continue
    else:
        ans.append(list[i])
        ed=list[i][1]
```

### 3 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（不同区间内含的点可以是同一个，位于区间端点上的点也算作区间内）。



这个题可以转化为上一题的求最大不相交区间的数量。【步骤一】：按照区间右端点从小到大排序。【步骤二】：从前往后依次枚举每个区间。假设当前遍历到的区间为第*i*个区间  $[l_i, r_i]$ ，有以下两种情况：【1】 $l_i < ed$ : 说明当前区间与前面区间有交集，前面已经选点了。因此直接跳过。【2】 $l_i > ed$ : 说明当前区间与前面没有交集。因此选中当前区间，并设置  $ed = r_i$ 。

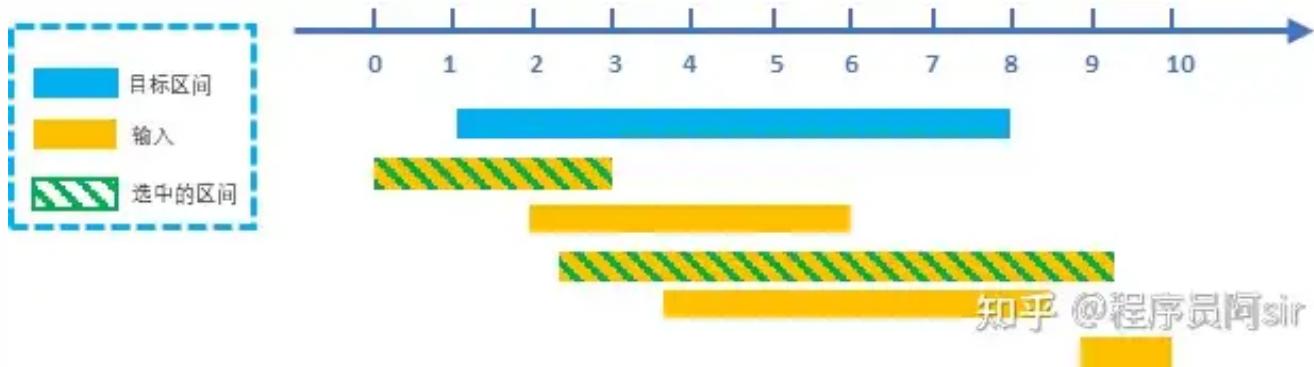
```

list.sort(key=lambda x:x[1])
ed=list[0][1]
ans=[]
for i in range(1,n):
    if list[i][0]<=ed:
        continue
    else:
        ans.append(list[i][1])
        ed=list[i][1]

```

## 4 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。



知乎 @程序员阿sir

**【步骤一】**：按照区间左端点从小到大排序。 **【步骤二】**：从前往后依次枚举每个区间，在所有能覆盖当前目标区间起始位置start的区间之中，选择右端点最大的区间。假设右端点最大的区间是第i个区间，右端点为r\_i。最后将目标区间的start更新成r\_i

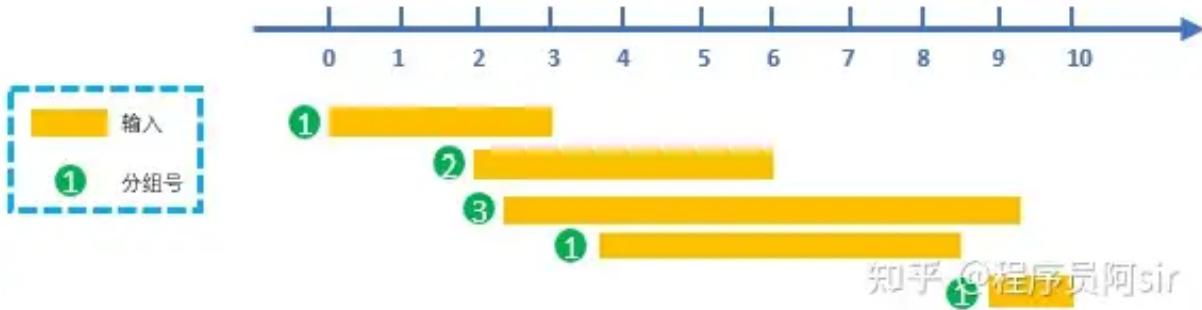
```

q.sort(key=lambda x:x[0])
start=0
end=0
ans=1
for i in range(n):
    if end==n-1:
        break
    if q[i][0]<=start<=q[i][1]:
        end=max(end,q[i][1])
    elif q[i][0]>start:
        ans+=1
        start=0
        start+=end

```

## 5 区间分组问题

给出一堆区间，问最少可以将这些区间分成多少组使得每个组内的区间互不相交。



【步骤一】：按照区间左端点从小到大排序。【步骤二】：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。

(即判断是否存在某个组的右端点在当前区间之中。如果可以，则不能放到这一组)

假设现在已经分了 m 组了，第 k 组最右边的一个点是  $r_k$ ，当前区间的范围是  $[L_i, R_i]$ 。则：

如果  $L_i < r_k$  则表示第 i 个区间无法放到第 k 组里面。反之，如果  $L_i > r_k$ ，则表示可以放到第 k 组。

- 如果所有 m 个组里面没有组可以接收当前区间，则当前区间新开一个组，并把自己放进去。
- 如果存在可以接收当前区间的组 k，则将当前区间放进去，并更新当前组的  $r_k = R_i$ 。

注意：

为了能快速的找到能够接收当前区间的组，我们可以使用优先队列（小顶堆）。

优先队列里面记录每个组的右端点值，每次可以在  $O(1)$  的时间拿到右端点中的的最小值。

```
import heapq
list.sort(key=lambda x: x[0])
min_heap = [list[0][1]]
for i in range(1, n):
    if list[i][0] > min_heap[0]:
        heapq.heappop(min_heap)
    heapq.heappush(min_heap, list[i][1])
num=len(min_heap)
```

#### 四、bfs

```
from collections import deque
n=int(input())
def inboard(x,y):
    return 0<=x<n and 0<=y<n
directions=[(-1,0),(1,0),(0,-1),(0,1)]
board=[list(map(int,input().split())) for _ in range(n)]
five=[]
for i in range(n):
    for j in range(n):
        if board[i][j]==9:
            end_x,end_y=i,j
```

```

    elif board[i][j]==5:
        five.append((i,j))
    (sx_1,sy_1),(sx_2,sy_2)=five[0],five[1]
def bfs(board,x_1,y_1,x_2,y_2,ex,ey):
    queue=deque([(x_1,y_1,x_2,y_2)])
    visited={(x_1,y_1,x_2,y_2)}
    while queue:
        cx_1,cy_1,cx_2,cy_2=queue.popleft()
        for dx,dy in directions:
            nx_1,ny_1,nx_2,ny_2=cx_1+dx,cy_1+dy,cx_2+dx,cy_2+dy
            if inboard(nx_1,ny_1) and inboard(nx_2,ny_2) and (nx_1,ny_1,nx_2,ny_2) not in
visited:
                visited.add((nx_1,ny_1,nx_2,ny_2))
                if (nx_1,ny_1)==(ex,ey) and board[nx_2][ny_2]!=1:#注意⚠️：即使一个格子碰到了终点，但另外一个格子碰到不可访问的格子，也不能输出'yes'
                    return 'yes'
                elif (nx_2,ny_2)==(ex,ey) and board[nx_1][ny_1]!=1:
                    return 'yes'
                elif board[nx_1][ny_1]!=1 and board[nx_2][ny_2]!=1:
                    queue.append((nx_1,ny_1,nx_2,ny_2))
    return 'no'
print(bfs(board,sx_1,sy_1,sx_2,sy_2,end_x,end_y))

```

1. 劣路径的剪枝：剪枝可以避免无效的路径计算，从而显著减少搜索空间。

```

if effort > min_effort[x][y]:
    continue#如果当前路径的累计代价 effort 已经大于记录的最优代价 min_effort[x][y]，则说明这条路径已经不是最优的，继续扩展它是没有意义的，直接跳过（剪枝）

```

2. 路径更新的剪枝：

```

if total_effort < min_effort[nx][ny]:
    min_effort[nx][ny] = total_effort
    heapq.heappush(pq, (total_effort, nx, ny))#只有当新路径的累计代价 total_effort 小于已知的代价 min_effort[nx][ny] 时，才更新邻居节点的代价并加入堆中

```

3. 起点或终点为阻碍的剪枝：如果起点或终点是不可通行的 (#)，直接输出 NO，不再进行路径搜索。

```

if terrain[sx][sy] == '#' or terrain[ex][ey] == '#':
    results.append('NO')

```

代码：

```

import heapq
def min_effort_dijkstra(terrain,m,n,start,end):
    directions=[(-1,0),(1,0),(0,-1),(0,1)]
    #使用 heapq 最小堆管理优先级队列，使得插入和取出操作的时间复杂度为 O(log n)，保证算法整体高效。
    pq=[]

```

```

heapq.heappush(pq,(0,start[0],start[1]))

min_effort=[[float('inf')]*n for _ in range(m)]
min_effort[start[0]][start[1]]=0

while pq:
    effort,x,y=heapq.heappop(pq)

    if(x,y)==end:
        return effort

    if effort>min_effort[x][y]:
        continue

    for dx,dy in directions:
        nx,ny=x+dx,y+dy

        if 0<=nx< m and 0<=ny< n and terrain[nx][ny]!='#':
            next_effort=abs(int(terrain[nx][ny])-int(terrain[x][y]))
            total_effort=effort+next_effort

            if total_effort<min_effort[nx][ny]:
                min_effort[nx][ny]=total_effort
                heapq.heappush(pq,(total_effort,nx,ny))

return 'NO'

m,n,p=map(int,input().split())
terrain=[input().strip().split() for _ in range(m)]

results=[]
for _ in range(p):
    sx,sy,ex,ey=map(int,input().split())
    if terrain[sx][sy]=='#' or terrain[ex][ey]=='#':
        results.append('NO')
    else:
        results.append(min_effort_dijkstra(terrain,m,n,(sx,sy),(ex,ey)))

for res in results:
    print(res)

```

```

import heapq
heapq.heapify
heapq.heappush(heap, item)#推入元素
heapq.heappop(heap)#弹出堆顶
heapq.heappushpop(heap, item)#先推再弹,更高效
heapq.heapreplace(heap, item)#先弹再推

```

```
from collections import deque
```

```

n=int(input())
board=[[int(x) for x in input()] for _ in range(n)]

directions=[(-1,0),(1,0),(0,1),(0,-1)]


def bfs_mark_island(x,y,mark):
    queue=deque([(x,y)])
    island=[]
    board[x][y]=mark
    while queue:
        cur_x,cur_y=queue.popleft()
        island.append((cur_x,cur_y))
        for dx,dy in directions:
            nx,ny=cur_x+dx,cur_y+dy
            if 0<=nx<n and 0<=ny<n and board[nx][ny]==1:
                board[nx][ny]=mark
                queue.append((nx,ny))
    return island

def bfs_shortest_distance(island1,island2):
    queue=deque()
    visited=set()
    for x,y in island1:
        queue.append((x,y,0))
        visited.add((x,y))

    while queue:
        cur_x,cur_y,cur_dist=queue.popleft()
        if (cur_x,cur_y) in island2:
            return cur_dist-1

        for dx,dy in directions:
            nx,ny=cur_x+dx,cur_y+dy
            if 0<=nx<n and 0<=ny<n and (nx,ny) not in visited:
                visited.add((nx,ny))
                queue.append((nx,ny,cur_dist+1))

islands=[]
mark=2
for i in range(n):
    for j in range(n):
        if board[i][j]==1:
            islands.append(bfs_mark_island(i,j,mark))
            mark+=1

island1,island2=islands
print(bfs_shortest_distance(island1,island2))

```

dfs

回溯

```

#马走日
directions=[[-2,1],[-1,2],[1,2],[2,1],[2,-1],[1,-2],[-1,-2],[-2,-1]]
def dfs(n,m,x,y,ma,step,p):
    if step==n*m:
        p[0]+=1
        return
    for dx,dy in directions:
        if -1<x+dx<n and -1<y+dy<m and ma[x+dx][y+dy]:
            ma[x+dx][y+dy]=0
            dfs(n,m,x+dx,y+dy,ma,step+1,p)
            ma[x+dx][y+dy]=1
case=int(input())
for _ in range(case):
    n,m,x,y=map(int,input().split())
    ma=[[0 for _ in range(m)] for _ in range(n)]
    ma[x][y]=1
    p=[0]
    dfs(n,m,x,y,ma,1,p)
    print(p[0])

#八皇后
def queens(index, prefix=[]):
    if len(prefix) == n:
        return [prefix]
    result = []
    for i in range(1, n + 1):
        if hashTable[i]:
            continue
        flag=True
        for row in range(1, index):
            if abs(index-row)==abs(i-prefix[row-1]):
                flag=False
                break
        if flag:
            hashTable[i] = True
            result +=queens(index+1, prefix + [i])
            hashTable[i] = False
    return result
n=8
hashTable = [False] *(n+1)
result = queens(1)
for _ in range(int(input())):
    t= int(input())
    print(''.join(map(str,result[t-1])))

```

```

#记忆性递归的优化
R,C=map(int,input().split())
top=[[0]*C]*R
middle=[[0]*C]+list(map(int,input().split()))+[0]*C for _ in range(R)]
board=top+middle+top
directions=[(0,1),(0,-1),(1,0),(-1,0)]

```

```

memo=[[0 for _ in range(C+2)]for _ in range(R+2)]
def max_area(board,memo):
    def dfs(board,x,y):
        if memo[x][y]!=0:#⚠️记忆性递归的关键
            return memo[x][y]
        if board[x][y]<min(board[x+1][y],board[x-1][y],board[x][y+1],board[x][y-1]):
            return 1
        else:
            count=1
            for dx,dy in directions:
                if board[x][y]>board[x+dx][y+dy]:
                    count=max(count,dfs(board,x+dx,y+dy)+1)
            memo[x][y]=count
            return count
    max_result=1
    for i in range(1,R+1):
        for j in range(1,C+1):
            if memo[i][j]!=0:
                current_dfs=memo[i][j]
            else:
                current_dfs=dfs(board,i,j)
            max_result=max(current_dfs,max_result)
    return max_result
print(max_area(board,memo))

```

```

# 有界的深度优先搜索
n, m, l = map(int, input().split())
graph = dict()
visited = [False] * n
for _ in range(m):
    s, e = map(int, input().split())
    if s not in graph.keys():
        graph[s] = [e]
    else:
        graph[s].append(e)
    if e not in graph.keys():
        graph[e] = [s]
    else:
        graph[e].append(s)

for i in graph:
    graph[i].sort()

route = []
def dfs(cur, depth):
    if visited[cur] or depth > l:
        return
    visited[cur] = True
    route.append(cur)

```

```

if cur not in graph.keys():
    return

for next in graph[cur]:
    dfs(next, depth+1)

start = int(input())
dfs(start, 0)
#print(' '.join([str(x) for x in route]))
print(*route)

```

## 迷宫问题

这是最经典的DFS问题，问能否走到出口、输出可行路径、输出连通分支数、输出连通块大小等。这种问题应用经典的图搜索DFS即可解决，不需要回溯，每个点只需要搜到一次，所以不需要撤销标记。

```

directions = [(-1,0),(1,0),(0,-1),(0,1)]
cnt = 0
ans = []
vis = [[False]*n for _ in range(m)]
def dfs(r,c):
    global area
    if vis[r][c]:
        return
    vis[r][c] = True
    area += 1
    for dr,dc in directions:
        if 0<=r+dr<m and 0<=c+dc<n and info[r+dr][c+dc]!='#':
            dfs(r+dr,c+dc)
for i in range(m):
    for j in range(n):
        if not vis[i][j]:
            cnt += 1
            area = 0
            dfs(i,j)
            ans.append(area)

```

注意这里要对下标越界与否做判断；另一种写法是在外面加一圈“#”作为保护圈，从而可以省去判断。

稍微麻烦一点的题目：OJ23558 有界的深度优先搜索，需要带step参数，在step=L时return

## DFS回溯（枚举）

这是DFS题目中最常出现的类型。由于要遍历所有可能，在每次做出选择之后需要撤销选择标记，以便不影响其它的路径（也就是说一个点会被搜到很多次）

这其实是一种枚举想法。理论上这些题目可以用一系列的for循环加条件判断解决，譬如八皇后问题，可以写7个for循环。但一方面这样写很麻烦，另一方面如果需要的循环数量n是变量，就没法这么写了。这时就要借助于DFS的思想。实质上DFS遍历的顺序和多重for循环的顺序是一致的，每次调用dfs函数的过程可以看成是下一层的for循环。

这种问题的实现细节其实有点麻烦。不同的写法可能都能成功，但一点微小的改动也可能对程序造成极大的影响，甚至return语句的不同顺序都可能导致不同的结果。关键在于何时做标记、撤销标记，何时返回，以及如何处理统计量。

下面以输出所有可能的最短路径为例展示回溯问题的写法。

```
vis = [[False]*n for _ in range(m)]
directions = [(-1,0),(1,0),(0,1),(0,-1)]
ans = []
dist = float('inf')
def dfs(r,c,pre=[],step=0):
    if step>dist:
        return
    if vis[r][c] or info[r][c]=='#':
        return
    vis[r][c] = True
    pre.append((r,c))
    if r==end_r and c==end_c:
        if step==dist:
            ans.append(pre[:]) #要加入的是pre的copy，因为pre会变
        return
    elif step < dist:
        ans = [pre[:]]
        dist = step
        return
    for dr,dc in directions:
        if 0<=r+dr<m and 0<=c+dc<n:
            dfs(r+dr,c+dc,pre,step+1) #也有在这里做选择和撤销选择的写法，也可以
            vis[r][c] = False
            pre.pop() #回溯，“撤销选择”
```

## 用栈实现DFS

由于递归的底层实现是栈，DFS算法基于递归，故用栈也能实现DFS（当然这样写的人比较少）

用栈实现DFS时和BFS的写法很像（每次取出一个点，将它的邻点中还未搜索过的点入栈），只是将BFS中的队列换成栈而已。从这个角度看，BFS和DFS之间似乎有一种“对偶”的关系。

如何用栈实现DFS回溯我还没想清楚，是否允许多次入栈即可？（入栈时打标记，出栈时撤销？这样的话就和spfa的写法很像了，但是没试过，不知道对不对）

## 六、数据结构

### 单调栈

```

#比它大的数在几位数后出现
p=list(map(int,input().split()))
index=[0]*len(p)
stack=[]
for i in range(len(p)-1,-1,-1):
    t=p[i]
    while stack and t>p[stack[-1]]:
        stack.pop()
    if stack:
        index[i]=stack[-1]-i
    stack.append(i)
print(index)

```

## 十六、Kadane算法求最大和子序列

```

def maxSubArray(arr):
    current_sum = 0
    max_sum = float('-inf') # 初始化为负无穷，处理全负数组的情况
    for num in arr:
        current_sum = max(num, current_sum + num) # 更新当前子数组和
        max_sum = max(max_sum, current_sum) # 更新最大子数组和
    return max_sum

```

这看起来很dp.若要存储路径(即最大子数组):

```

def maxSubArray(arr):
    current_sum = 0
    max_sum = float('-inf')
    start = 0 # 当前子数组的起始索引
    end = 0 # 最大子数组的结束索引
    temp_start = 0 # 临时存储起始索引
    for i, num in enumerate(arr):
        if num > current_sum + num: # if.....else状态转移
            current_sum = num
            temp_start = i # 重新开始新的子数组
        else:
            current_sum += num
        if current_sum > max_sum:
            max_sum = current_sum
            start = temp_start # 更新最大子数组的起始索引
            end = i # 更新最大子数组的结束索引
    return max_sum, arr[start:end+1]

```

## 二维数组扩展

```
def maxSubMatrix(matrix):
    if not matrix or not matrix[0]:
        return 0
    rows, cols = len(matrix), len(matrix[0])
    max_sum = float('-inf')
    for top in range(rows): # 遍历子矩阵上边界
        row_sum = [0] * cols # 对每一次上边界, 初始化列和
        for bottom in range(top, rows): # 从top到rows行之间的列和
            for col in range(cols):
                row_sum[col] += matrix[bottom][col]
            current_sum = 0 # 使用 Kadane's 算法求当前列和的最大值
            # 求列和的最大子数组和, 并更新全局最大和
            local_max = float('-inf')
            for x in row_sum:
                current_sum = max(x, current_sum + x)
                local_max = max(local_max, current_sum)
            max_sum = max(max_sum, local_max)
    return max_sum
```

## 一维周期数组(环形数组)扩展

```
def maxCircularSubArray(arr):
    max_kadane = kadane(arr) # 普通 Kadane 最大子数组和
    total_sum = sum(arr)
    min_kadane = kadane([-x for x in arr]) # (负的)最小子数组和
    max_circular = total_sum + min_kadane # 环形子数组和
    # 如果全是负数, max_circular 会变成 0, 所以只返回 max_kadane
    return max(max_kadane, max_circular) # 两种可能情况取max
def kadane(arr):
    current_sum = 0
    max_sum = float('-inf')
    for num in arr:
        current_sum = max(num, current_sum + num)
        max_sum = max(max_sum, current_sum)
    return max_sum
```