

Input and Output

输入输出

读取文件

持续输入

List and Tuple

插入

解包

二维数组

`range` 负步长

切片

遍历

排序

赋值, 浅拷贝, 深拷贝

递归

面向对象

数据结构

`dict`

BFS ( `deque` 用法)

Dijkstra ( `heap` 用法)

## 📌 Important

例如：1)递归是数算中必备的核心技能, 建议优先掌握, 可以参看 [https://github.com/GMyhf/2024fall-cs101/blob/main/20241029\\_recursion.md](https://github.com/GMyhf/2024fall-cs101/blob/main/20241029_recursion.md) 2)队列在广度优先搜索(BFS)中有着广泛的应用. 其他班级可能还没有讲搜索, 可以参看, [https://github.com/GMyhf/2024fall-cs101/blob/main/20241119\\_searching.md](https://github.com/GMyhf/2024fall-cs101/blob/main/20241119_searching.md) 3)其他的常用技巧, 没学过也没关系, 遇到相关题目时逐一掌握即可. 如: 双指针(链表里有个 快慢指针需要掌握), 单调栈, 二分查找, 并查集, 滑动窗口, 懒删除等. 通过 1~2 道题即可理解基础原理, 但要熟练掌握需要多加练习. 4) OOP 的写法属于语法范畴, 可以通过阅读文档快速掌握. <https://www.runoob.com/python3/python3-class.html>

数算的学习, 也是一方面学习原理, 手搓数据结构和算法实现, 另一方面做题时候直接使用现有包, 如 stack, deque, heapq, sort, permutation等. 编程平台通常是python解释器的基础版(没有额外的包. 可喜的是看到 洛谷支持numpy), 不支持的数据结构和算法需要自己代码实现.

2025/2/3 说明: 如果你已经完成了 LeetCode 热题 100, <https://leetcode.cn/studyplan/top-100-linked/>, 那么接下来可以继续 面试经典150题, <https://leetcode.cn/studyplan/top-interview-150/>. 你会惊喜的发现其中有一半做好了, 因为这两套题目之间存在很大的重叠

<https://github.com/javasmall/bigsai-algorithm/tree/master>

## 📌 Important

PEP 8 - Style Guide for Python Code <https://peps.python.org/pep-0008/>

PEP是“Python Enhancement Proposal”的缩写, 意为“Python增强提案”. 其中最著名的可能是PEP 8, 它是Python代码风格指南, 为编写清晰一致的Python代码提供了指导原则.

在Python编程中, 命名规范对于代码的可读性和维护性至关重要. 遵循一致的命名规则可以使代码更易于理解, 也便于团队协作. 以下是Python中常用的命名规范:

### 类名

- 使用大写字母开头的单词(即PascalCase), 例如: `MyClass`, `UserProfile`.
- 避免使用下划线.

### 函数名

- 应该使用小写字母, 单词之间用下划线分隔(即snake\_case), 如: `my_function`, `calculate_area`.

### 变量名

- 与函数名一样, 变量名也应该使用小写字母, 单词间用下划线连接(snake\_case), 例如: `user_name`, `total_value`.
- 对于临时或短生命周期的变量, 可以使用简短的名字, 比如: `i`, `j`, `x`.

这些规则并非强制, 但在大多数情况下遵循PEP 8(Python的官方风格指南)中的建议会使你的代码更加专业和易于理解.

## 考前准备清单:

1. pycharm 提前开好六个文件和一个 `input` 文件
2. 开启 `win+v` 剪切板

3. 

```
1 import sys
2 sys.stdin = open("input", "r")
```
4. `int(input())`, `map(int, input().split())`, `for i in range(n)`
5. Tree 模板, DisjointSet 模板包括 `parents`
6. 头文件: `from collections import deque, defaultdict`  
`from typing import List, Dict, Set`  
`from heapq import heappop, heappush, heapify`

## Input and Output

### 输入输出

- 输入整数 `n = int(input())`, 默认输入为字符串, 需转为整数  
输入多个整数 `m, n = map(int, input().split())` (`split()` 按空格将字符分段)  
输入数组 `num = list(map(int, input().split()))`
- f-string  
输出 `print(f"The answer is {m} {n}")`  
保留两位小数 `result = f"{num:.2f}"`

```
1 name = "Alice"
2 print(f"{name:<10}") # Left align (width 10) : "Alice    "
3 print(f"{name:>10}") # Right align (width 10): "      Alice"
```

- `str` 转为 `chr` 数组 `s_list = list(s)`
- 将 `str` 数组连接成 `str` `print(''.join(s_list))`  
数组输出 `print(' '.join(map(str, num)))` 或者 `print(*num, sep=' ')`
- 无换行输出 `print(n, end = '')`
- 输入字符串 `s = input()`
- 无穷大和无穷小: `float("inf")` 和 `float("-inf")`  
或者 `from math import inf`
- ASCII 码 `ord("A")` 和 `chr(65)`

### 读取文件

```
1 import sys
2
3 sys.stdin = open('input.txt', 'r')
4 sys.stdout = open('output.txt', 'w')
```

## 持续输入

```
1 import sys
2
3 for line in sys.stdin:
4     s = line.strip()
5     pass
```

或者

```
1 while 1:
2     try:
3         s = input()
4         pass
5     except EOFError:
6         break
7     # except : break
```

## List and Tuple

插入

```
l.append(num)
```

解包

```
1 x, y, z = [1, 2, 3]
2 print(x, y, z)
3
4 x, y, z = (4, 5, 6)
5 print(x, y, z)
6
7 (x, y), z = ((1, 2), 3)
8 print(x, y, z)
9
10 x, *y, z = [1, 2, 3, 4, 5]
11 print(x, y, z) # output: 1 [2, 3, 4] 5
12
13 a, b = b, a #交换元素
```

二维数组

```
DP = [[0]*n for _ in range(n + 1)]
```

```
1 r, c = map(int, input().split())
2 matrix = [list(map(int, input().split())) for _ in range(r)]
```

```
1 # 加保护圈
2 maze = []
3 maze.append( [-1 for x in range(m+2)] )
4 for _ in range(n):
5     maze.append([-1] + [int(_) for _ in input().split()] + [-1])
6 maze.append( [-1 for x in range(m+2)] )
```

## range 负步长

`range(start, stop, step)`

```
1 for i in range(10, 0, -2):
2     print(i, end = "")
3 # output: 10, 8, 6, 4, 2
```

## 切片

规则同 `range` 的三个参数, 分别为 `start`, `end`, `step`

```
1 l = ['I', 'l', 'o', 'v', 'e', 'p', 'y', 't', 'h', 'o', 'n']
2 print(l[3:]) # print the 3th~the last element : ['v', 'e', 'p', 'y', 't', 'h', 'o', 'n']
3 print(l[-1]) # print the last element : n
4 print(l[:2]) # output : ['I', 'o', 'e', 'y', 'h', 'n']
```

## 遍历

```
1 l = ['I', 'love', 'python']
2 for i in l:
3     print(i, end=' ') # I love python
```

```
1 for i, val in enumerate(nums, start = 1)
```

## 排序

```
1 triplets = [(1, 5, 3), (2, 1, 4), (3, 7, 2), (4, 3, 6)]
2 sorted_triplets = sorted(triplets, key=lambda x: x[1], reverse=True) # 排序是基于每个三元组的第二个元素
3 # 默认升序, 加 reverse 变成降序 : [(3, 7, 2), (1, 5, 3), (4, 3, 6), (2, 1, 4)]
```

# 赋值, 浅拷贝, 深拷贝

不可变对象包括: `int`, `float`, `str`, 元组 `tuple` 以及自定义的一些 `class` 如 `ListNode`, `TreeNode`

- 内容一旦创建后就无法修改. 对象的修改会创建一个新的对象, 而不会修改原有对象.
- 赋值, 修改, 切片等操作都会返回新的对象.

可变对象包括: 列表 `list`, 字典 `dict`, 集合 `set`

- 修改可变对象时, 原有对象的内容会被改变.
- 对可变对象的操作(如增加, 删除元素)会影响到所有引用该对象的变量.

```
1. 1 nums = [[]] * 3 # 指向同一个 list
   2     nums[0].append(1)
   3     print(nums) # output : [[1], [1], [1]]
```

```

2. 1 a = [1, 2, [3, 4]]
    2 b = a # 赋值, b, a指向同一块内存
    3
    4 b[0] = 10
    5 b[2][0] = 100 # 将这一块内存的数据修改
    6
    7 print(a) # output: [10, 2, [100, 4]]
    8 print(b) # output: [10, 2, [100, 4]]
    9 #修改影响原对象

```

### 3. 浅复制

```

1 nums = [1, 2, [3, 4]]
2 shallow_copy = nums[:] # 浅复制, 但a[2]和b[2]仍指向同一个子列表[3, 4].
3 # shallow_copy = nums.copy()
4 # shallow_copy = list(nums)
5
6 shallow_copy[0] = 0
7 shallow_copy[2][0] = 0
8
9 print(nums) # output : [1, 2, [0, 4]]
10 print(shallow_copy) # output : [0, 2, [0, 4]]
11 # 这些浅复制方法会创建一个新的列表对象
12 # 如果列表中的元素是可变对象 (比如list, dict等), 这些元素本身仍然会被共享(即引用相同的内存地址)
13 # 如果列表中的元素是不可变对象 (比如int, str等), 新列表中的元素与原始列表中的元素不会相互影响.

```

#### 应用:

```

1 def add(nums):
2     nums.append(0)
3     return nums
4
5 nums = [1,2,3,4,5]
6 nums1 = add(nums)
7 print(nums) # output : [1, 2, 3, 4, 5, 0]
8
9 nums = [1,2,3,4,5]
10 nums1 = add(nums.copy())
11 print(nums) # output : [1, 2, 3, 4, 5]

```

### 4. 深复制

```

1 import copy
2
3 nums = [1, 2, [3, 4]]
4 deep_copy = copy.deepcopy(nums) # 深拷贝, 创建完全独立的副本
5
6 deep_copy[0] = 0
7 deep_copy[2][0] = 0
8 print(nums) # output : [1, 2, [3, 4]]
9 print(deep_copy) # output : [0, 2, [0, 4]]

```

## 5. 原地修改

```
1 def rotate(nums, k):
2     nums[:] = nums[-k:] + nums[:-k] # 原地修改
3     # nums 作为参数传递到函数内时，函数内的 nums 是对原始 nums 列表的引用。
4
5     nums = [1,2,3,4,5]
6     rotate(nums, 2)
7     print(nums) # output : [4,5,1,2,3]
```

```
1 def rotate(nums, k):
2     nums = nums[-k:] + nums[:-k]
3
4     nums = [1,2,3,4,5]
5     rotate(nums, 2)
6     print(nums) # output : [1,2,3,4,5]
```

## 6. global

```
1 x = 10 # 全局变量
2
3 def modify_global():
4     global x # 使用 global 关键字，表示修改全局变量 x
5     x = 20
6
7 modify_global()
8 print(x) # output : 20
```

## 7. nonlocal

```
1 class Solution:
2     def outer():
3         x = 10 # 外层函数的局部变量
4         def inner():
5             nonlocal x # 使用 nonlocal 关键字，表示修改外层函数的变量 x
6             x = 20
7         inner()
8         print(x) # output : 20
9
10    outer()
```

# 递归

```
1 from functools import lru_cache
2
3 @lru_cache(maxsize=None)
4 def fun():
5     .....
```

# 面向对象

- ```
1  from math import gcd
2
3
4  class frac():
5      def __init__(self, a, b):
6          self.a = a
7          self.b = b
8
9      def __add__(self, other):
10         na = self.a * other.b + self.b * other.a
11         nb = self.b * other.b
12         return frac(na // gcd(na, nb), nb // gcd(na, nb))
13
14     def __eq__(self, other):
15         return self.num * other.den == other.num * self.den
16
17     def __lt__(self, other):
18         return self.a * other.b < self.b * other.a
19
20     def __str__(self):
21         return f"{self.a}/{self.b}"
22
23     def show(self):
24         print(f"{self.a}/{self.b}")
25
26
27 def main():
28     fractions = [frac(1, 2), frac(1, 3), frac(1, 1)]
29     fractions = sorted(fractions) # [frac(1, 1), frac(1, 2), frac(1, 3)]
30
31     a, b, c, d = map(int, input().split())
32     print(frac(a, b) + frac(c, d))
33     (frac(a, b) + frac(c, d)).show()
34
35 if __name__ == "__main__":
36     main()
```

`__lt__` 应用: 可以实现类似c++中 `sort()` 函数 `cmp` 的功能!!! [OpenJudge - 07618:病人排队](#)

- Python类中的方法调用

```
1  class solution:
2      def method1(self):
3          print("This is method1")
4
5      def method2(self):
6          print("This is method2")
7          self.method1()
```



# 数据结构

|     | deque                                      | heapq                                                     | stack                                   | set                                               |
|-----|--------------------------------------------|-----------------------------------------------------------|-----------------------------------------|---------------------------------------------------|
| 库   | <code>from collections import deque</code> | <code>from heapq import heapify, heappop, heappush</code> | -                                       | -                                                 |
| 定义  | <code>dq = deque()</code>                  | <code>que = heapify[nums]</code>                          | <code>st = []</code>                    | <code>s = set()</code>                            |
| 初始化 | <code>dq = deque([0,1,2])</code>           |                                                           |                                         | <code>s = set([1])</code> 或者 <code>s = {1}</code> |
| 入   | <b>入队</b><br><code>dq.append(num)</code>   | <code>heappush(que, num)</code>                           | <b>入栈</b><br><code>s.append(num)</code> | <b>添加</b> <code>s.add(num)</code>                 |
| 出   | <b>出队</b><br><code>dq.popleft()</code>     | <code>heappop(que)</code>                                 | <b>出栈</b> <code>s.pop()</code>          | <b>删除</b><br><code>s.discard(num)</code>          |
| 非空  | <code>if deque</code>                      | <code>if que</code>                                       | <code>if st</code>                      | <code>if s</code>                                 |

各种 pop 使用前一定要检查是否为空

## dict

```
1 my_dict = {'a': 1, 'b': 2, 'c': 3}
2 my_dict['d'] = 4 # 添加元素
3
4 print('a' in my_dict) # 访问key
5 print(2 in my_dict.values()) # 访问value
6
7 for key, value in my_dict.items(): # 遍历键值对
8     print(key, value)
9
10 value = my_dict.pop('b') # 删除键 'b', 并返回它对应的值
11 print(value, my_dict) # output: 2 {'a': 1, 'c': 3, 'd': 4}
12
13 # 使用 get(key, default=None) 方法时, 如果 key 存在, 获取对应的 value 值
14 # 如果 key 不存在, 不会抛出 KeyError 异常, 返回 default (default 默认为 None)
15 my_dict = {'a': 1, 'c': 3, 'd': 4}
16 print(my_dict.get('a')) # output : 1
17 print(my_dict.get('b')) # output : None
18 print(my_dict.get('b', 'not found')) # output : not found
19
20 #.setdefault(key, default=None) 方法, 如果 key 存在, 获取对应的 value 值
21 # 如果 key 不存在, 将 key 添加到字典中, 并将其 value 设为 default (default 默认为 None) 并返回 default
22 my_dict = {'a': 1, 'c': 3, 'd': 4}
23 res = my_dict.setdefault('a', 10)
24 print(res, my_dict) # 1 {'a': 1, 'c': 3, 'd': 4}
```

```

25 res = my_dict.setdefault('b', 2)
26 print(res, my_dict) # 2 {'a': 1, 'c': 3, 'd': 4, 'b': 2}

```

## defaultdict

```

1 from collections import defaultdict
2
3 # 如果键不存在时, 默认值是 int(), 即 0
4 d = defaultdict(int)
5 print(d['a']) # output : 0, 因为 'a' 键没有在字典中, 默认值为 int() 即 0
6
7 # 如果键不存在时, 默认值是 list(), 即一个空列表
8 d = defaultdict(list)
9 d['a'].append(1)
10 print(d) # output : defaultdict(<class 'list'>, {'a': [1]})
11
12 # 如果键不存在时, 默认值是 set(), 即一个空集合
13 d = defaultdict(set)
14 d['a'].add(1)
15 print(d) # output : defaultdict(<class 'set'>, {'a': {1}})

```

## set

- 添加多个元素(可以传入列表, 元组, 其他集合等)

```

1 my_set = {1, 2, 3, 4}
2 my_set.update([4, 5, 6])
3 print(my_set) # output: {1, 2, 3, 4, 5, 6}
4
5 my_set.discard(4)
6 print(my_set) # output : {1, 2, 3, 5, 6}
7 my_set.discard(7)
8 print(my_set) # output : {1, 2, 3, 5, 6}

```

- 数学运算

```

1 set_a = {1, 2, 3}
2 set_b = {2, 3, 4}
3 print(set_a & set_b) #交集 output :{2, 3}
4 print(set_a | set_b) #并集 output :{1, 2, 3, 4}
5 print(set_a - set_b) #差集 output :{1}
6 print(set_a ^ set_b) #对称差集 output :{1, 4}

```

## BFS (deque用法)

```

1 from collections import deque
2
3 dir = [(-1, 0), (0, 1), (1, 0), (0, -1)]
4
5 def isvalid(nx, ny):
6     return nx in range(1, n + 1) and ny in range(1, m + 1)
7
8 n, m, x, y = map(int, input().split())

```

```

9  Steps = [[-1] * (m + 1) for _ in range(n + 1)]
10
11  que = deque([(x, y, 0)])
12  while que:
13      px, py, step = que.popleft()
14      Steps[px][py] = step
15      for dx, dy in dir:
16          nx, ny = px + dx, py + dy
17          if isValid(nx, ny) and Steps[nx][ny] == -1:
18              Steps[nx][ny] = step + 1
19              que.append((nx, ny, step + 1))
20
21  for i in range(1, n + 1):
22      print(' '.join(map(str, Steps[i][1 : ])))

```

## Dijkstra (heap用法)

[OpenJudge - 20106:走山路](#)

```

1  from heapq import heappop, heappush
2
3  dir = [[-1, 0], [0, 1], [1, 0], [0, -1]]
4
5  def isValid(x, y):
6      return x in range(m) and y in range(n) and graph[x][y] != '#'
7
8  def search(sx, sy, ex, ey):
9      global graph
10     if graph[sx][sy] == '#' or graph[ex][ey] == '#':
11         return "NO"
12     que = [(0, sx, sy)]
13     visited = {(sx, sy)}
14     while que:
15         sum, px, py = heappop(que)
16         visited.add((px, py))
17         if px == ex and py == ey : return sum
18         for k in range(4):
19             nx, ny = px + dir[k][0], py + dir[k][1]
20             if isValid(nx, ny) and (nx, ny) not in visited:
21                 nsum = sum + abs(int(graph[nx][ny]) - int(graph[px][py]))
22                 heappush(que, (nsum, nx, ny))
23     return "NO"
24
25  m, n, p = map(int, input().split())
26  graph = [list(input().split()) for _ in range(m)]
27  for _ in range(p):
28      sx, sy, ex, ey = map(int, input().split())
29      print(search(sx, sy, ex, ey))

```