> **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 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 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-liked/, 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.

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}")`
- 数组输出 `print(' '.join(map(str, num)))` 或者 `print(*num, sep=' ')`
- 无换行输出 `print(n, end = '')`
- 保留两位小数 `result = f"{num:.2f}"`
- 输入字符串 `s = input()`
- 转为char数组 `s_list = list(s)`
- 输出char数组 `print(''.join(s_list))`
- 无穷大和无穷小： `float("inf")` 和 `float("-inf")`
- ASCII 码 `ord("A")` 和 `chr(65)`

读取文件

```
import sys

sys.stdin = open('input.txt', 'r')
sys.stdout = open('output.txt', 'w')
```

持续输入

```
import sys

for line in sys.stdin:
    s = line.strip()
    pass
```

或者

```
while 1:
    try:
        s = input()
        pass
    except EOFError:
        break
    # except : break
```

List and Tuple

插入

`l.append(num)`

解包

```
x, y, z = [1, 2, 3]
print(x, y, z)

x, y, z = (4, 5, 6)
print(x, y, z)

(x, y), z = ((1, 2), 3)
print(x, y, z)

x, *y, z = [1, 2, 3, 4, 5]
print(x, y, z)  # output: 1 [2, 3, 4] 5

a, b = b, a #交换元素
```

二维数组

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

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

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

`range` 负步长

`range(start, stop, step)`

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

切片

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

遍历

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

```
for index ele in enumerate(num, start = 1)
```

排序

```
triplets = [(1, 5, 3), (2, 1, 4), (3, 7, 2), (4, 3, 6)]
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
# 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. ```
   nums = [[]] * 3 # 指向同一个 list
   nums[0].append(1)
   print(nums) #  output : [[1], [1], [1]]
   ```

2. ```
   a = [1, 2, [3, 4]]
   b = a # 赋值, b, a指向同一块内存

   b[0] = 10
   b[2][0] = 100 # 将这一块内存的数据修改

   print(a)  # output: [10, 2, [100, 4]]
   print(b)  # output: [10, 2, [100, 4]]
   #修改影响原对象
   ```

3. **浅复制**

```python
nums = [1, 2, [3, 4]]
shallow_copy = nums[:] # 浅复制, 但a[2]和b[2]仍指向同一个子列表[3, 4].
# shallow_copy = nums.copy()
# shallow_copy = list(nums)

shallow_copy[0] = 0
shallow_copy[2][0] = 0

print(nums)         # output : [1, 2, [0, 4]]
print(shallow_copy) # output : [0, 2, [0, 4]]
# 这些浅复制方法会创建一个新的列表对象
# 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)
# 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：

```python
def add(nums):
    nums.append(0)
    return nums

nums = [1,2,3,4,5]
nums1 = add(nums)
print(nums) # output : [1, 2, 3, 4, 5, 0]

nums = [1,2,3,4,5]
nums1 = add(nums.copy)
print(nums) # output : [1, 2, 3, 4, 5]
```

4. **深复制**

```python
import copy

nums = [1, 2, [3, 4]]
deep_copy = copy.deepcopy(nums) # 深拷贝, 创建完全独立的副本

deep_copy[0] = 0
deep_copy[2][0] = 0
print(nums)           # output : [1, 2, [3, 4]]
print(deep_copy)      # output : [0, 2, [0, 4]]
```

5. 原地修改

```python
def rotate(nums, k):
    nums[:] = nums[-k:] + nums[:-k] # 原地修改
    # 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.

nums = [1,2,3,4,5]
rotate(nums, 2)
print(nums) # output : [4,5,1,2,3]
```

```python
def rotate(nums, k):
    nums = nums[-k:] + nums[:-k]

nums = [1,2,3,4,5]
rotate(nums, 2)
print(nums) # output : [1,2,3,4,5]
```

6. `global`

```python
x = 10  # 全局变量

def modify_global():
    global x  # 使用 global 关键字，表示修改全局变量 x
    x = 20

modify_global()
print(x)  # output : 20
```

7. `nonlocal`

```python
class Solution:
    def outer():
        x = 10  # 外层函数的局部变量
        def inner():
            nonlocal x  # 使用 nonlocal 关键字，表示修改外层函数的变量 x
            x = 20
        inner()
        print(x)  # output : 20

outer()
```

递归

```python
from functools import lru_cache

@lru_cache(maxsize=None)
def fun():
    ......
```

面向对象

- ```python
  from math import gcd


  class frac():
      def __init__(self, a, b):
          self.a = a
          self.b = b

      def __add__(self, other):
          na = self.a * other.b + self.b * other.a
  ```
  ```

```
        nb = self.b * other.b
        return frac(na // gcd(na, nb), nb // gcd(na, nb))

    def __eq__(self, other):
        return self.num * other.den == other.num * self.den

    def __lt__(self, other):
        return self.a * other.b < self.b * other.a

    def __str__(self):
        return f"{self.a}/{self.b}"

    def show(self):
        print(f"{self.a}/{self.b}")


def main():
    fractions = [frac(1, 2), frac(1, 3), frac(1, 1)]
    fractions = sorted(fractions) # [frac(1, 1), frac(1, 2), frac(1, 3)]

    a, b, c, d = map(int, input().split())
    print(frac(a, b) + frac(c, d))
    (frac(a, b) + frac(c, d)).show()

if __name__ == "__main__":
    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**

```
class Solution:
    def method1(self):
        print("This is method1")

    def method2(self):
        print("This is method2")
        self.method1()
```

数据结构

	deque	heapq	stack	set
库	`from collections import deque`	`from heapq import heapify, heappop, heappush`	-	-
定义	`dq = deque()`	`que = heapify[nums]`	`st = []`	`s = set()`
初始化	`dq = deque([0,1,2])`			`s = set([1])` 或者 `s = {1}`

	deque	heapq	stack	set
入	**入队** `dq.append(num)`	`heappush(que, num)`	**入栈** `s.append(num)`	**添加** `s.add(num)`
出	**出队** `dq.popleft()`	`heappop(que)`	**出栈** `s.pop()`	**删除** `s.disgard(num)`
非空	`if deque`	`if que`	`if st`	`if s`

各种 `pop` 使用前一定要检查是否为空

`dict`

```python
my_dict = {'a': 1, 'b': 2, 'c': 3}
my_dict['d'] = 4 # 添加元素

print('a' in my_dict) # 访问key
print(2 in my_dict.values()) # 访问value

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

value = my_dict.pop('b')  # 删除键 'b', 并返回它对应的值
print(value, my_dict)  # output: 2 {'a': 1, 'c': 3, 'd': 4}

# 使用 get(key, default=None) 方法时, 如果 key 存在, 获取对应的 value 值
# 如果 key 不存在, 不会抛出 KeyError 异常, 返回 default (default 默认为 None)
my_dict = {'a': 1, 'c': 3, 'd': 4}
print(my_dict.get('a')) # output : 1
print(my_dict.get('b')) # output : None
print(my_dict.get('b', 'not found')) # output : not found

# setdefault(key, default=None) 方法, 如果 key 存在, 获取对应的 value 值
# 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
my_dict = {'a': 1, 'c': 3, 'd': 4}
res = my_dict.setdefault('a', 10)
print(res, my_dict) # 1 {'a': 1, 'c': 3, 'd': 4}
res = my_dict.setdefault('b', 2)
print(res, my_dict) # 2 {'a': 1, 'c': 3, 'd': 4, 'b': 2}
```

`defaultdict`

```python
from collections import defaultdict

# 如果键不存在时, 默认值是 int(), 即 0
d = defaultdict(int)
print(d['a'])  # output : 0, 因为 'a' 键没有在字典中, 默认值为 int() 即 0

# 如果键不存在时, 默认值是 list(), 即一个空列表
d = defaultdict(list)
d['a'].append(1)
```

```
print(d)  # output : defaultdict(<class 'list'>, {'a': [1]})

# 如果键不存在时，默认值是 set()，即一个空集合
d = defaultdict(set)
d['a'].add(1)
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)

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

my_set.discard(4)
print(my_set) # output : {1, 2, 3, 5, 6}
my_set.discard(7)
print(my_set) # output : {1, 2, 3, 5, 6}
```

- 数学运算

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

BFS (`deque`用法)

```
from collections import deque

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

def isValid(nx, ny):
    return nx in range(1, n + 1) and ny in range(1, m + 1)

n, m, x, y = map(int, input().split())
Steps = [[-1] * (m + 1) for _ in range(n + 1)]

que = deque([(x, y, 0)])
while que:
    px, py, step = que.popleft()
    Steps[px][py] = step
    for dx, dy in dir:
        nx, ny = px + dx, py + dy
        if isValid(nx, ny) and Steps[nx][ny] == -1:
            Steps[nx][ny] = step + 1
            que.append((nx, ny, step + 1))

for i in range(1, n + 1):
    print(' '.join(map(str,Steps[i][1 : ])))
```

Dijstra (heap用法)

OpenJudge - 20106:走山路

```python
from heapq import heappop, heappush

dir = [[-1, 0], [0, 1], [1, 0], [0, -1]]

def isValid(x, y):
    return x in range(m) and y in range(n) and graph[x][y] != '#'

def search(sx, sy, ex, ey):
    global graph
    if graph[sx][sy] == '#' or graph[ex][ey] == '#':
        return "NO"
    que = [(0, sx, sy)]
    visited = {(sx, sy)}
    while que:
        sum, px, py = heappop(que)
        visited.add((px, py))
        if px == ex and py == ey : return sum
        for k in range(4):
            nx, ny = px + dir[k][0], py + dir[k][1]
            if isValid(nx, ny) and (nx, ny) not in visited:
                nsum = sum + abs(int(graph[nx][ny]) - int(graph[px][py]))
                heappush(que, (nsum, nx, ny))
    return "NO"

m, n, p = map(int, input().split())
graph = [list(input().split()) for _ in range(m)]
for _ in range(p):
    sx, sy, ex, ey = map(int, input().split())
    print(search(sx, sy, ex, ey))
```