基础知识点：

格式化输出
a=1/3 print(f'{a:.2f}')保留两个小数点

number = 42
print(f"{number:d}") # 输出 42（整数）

pi = 3.141592653589793
print(f"{pi:.2f}") # 输出 3.14，保留两位小数

accuracy = 0.856
print(f"{accuracy:.2%}") # 输出 85.60%，将浮动数转换为百分比

6. 进制转换（整数的二进制、八进制、十六进制表示）

Python 支持将整数转换为二进制、八进制和十六进制。

示例：

```python
number = 255
print(f"{number:b}")  # 输出二进制 '11111111'
print(f"{number:o}")  # 输出八进制 '377'
print(f"{number:x}")  # 输出十六进制 'ff'
```

- `map()`：将指定函数映射到可迭代对象的每个元素

```python
result = map(str, [1, 2, 3])  # 将每个整数转为字符串
list(result)  # 返回 ['1', '2', '3']
```

- `filter()`：过滤掉不满足条件的元素

```python
result = filter(lambda x: x > 1, [1, 2, 3])  # 过滤掉小于等于1的元素
list(result)  # 返回 [2, 3]
```

- `zip()`：将多个可迭代对象“打包”成元组

```python
list(zip([1, 2], ['a', 'b']))  # 返回 [(1, 'a'), (2, 'b')]
```

字符串操作：

- `split()`：将字符串分割为列表

```python
"a,b,c".split(",")  # 返回 ['a', 'b', 'c']
```

- `join()`：将列表连接成字符串

```python
",".join(['a', 'b', 'c'])  # 返回 'a,b,c'
```

lower()和upper（）和swapcase()：转换大小写

- `replace()`：替换字符串中的内容

```python
"hello world".replace("world", "Python")  # 返回 'hello Python'
```

- `find()`：查找子字符串的位置，找不到返回 -1

```python
"hello".find("l")  # 返回 2
```

”Hello“.lower() #返回‘hello’

s_swapcase=s.swapcase()

列表操作

- `append()`：在列表末尾添加元素

```python
lst = [1, 2]
lst.append(3)  # lst 变为 [1, 2, 3]
```

- `extend()`：将一个列表的元素添加到另一个列表

```python
lst = [1, 2]
lst.extend([3, 4])  # lst 变为 [1, 2, 3, 4]
```

- `pop()`：弹出并返回列表中的最后一个元素

```python
lst = [1, 2, 3]
lst.pop()  # 返回 3，lst 变为 [1, 2]
```

- `remove()`：删除列表中的某个元素

```python
lst = [1, 2, 3]
lst.remove(2)  # lst 变为 [1, 3]
```

- `sort()` **和** `reverse()`：排序和反转列表

```python
lst = [3, 1, 2]
lst.sort()  # lst 变为 [1, 2, 3]
lst.reverse()  # lst 变为 [3, 2, 1]
```

Math 的用法

1.基本数学运算

math.sqrt（x）返回 x 的平方根

math.trunc(x) 返回 x 的整数部分

math.ceil(x) 向上取整

math.floor(x) 向下取整

math.gcd(x,y) 返回 x 和 y 的最大公约数

2.阶乘与组合、排列

math.comb(n,k)

从 n 个元素中选出 k 个元素进行组合

math.perm(n,k)

从 n 个元素中选取 k 个元素来进行排列

math.factorial(n)

返回 n 的阶乘

3.返回整数平方根

math.isqrt(17)　　4

math.isqrt(16)　　4

递归，回溯

递归：

特点；从大不断往小，最后到达基本条件得到返回值

递归三法则：

1.递归算法必须有一个基本情况

2.递归算法必须改变其状态并不断朝基准情形前进

3.递归算法必须调用自身

基准情形是允许算法停止递归的条件。基准情形通常是一个足够小可以直接解决的问题。在 vectsum 算法中，基准情形是一个长度为 1 的列表。

例3：子集问题（回溯）

给定一个集合 `{1, 2, 3}`，输出所有子集。

python

```python
def subsets(nums):
    result = []

    def backtrack(start, current_subset):
        # 终止条件：如果当前子集包含所有元素
        result.append(list(current_subset))  # 保存当前子集

        # 递归选择：从 start 开始，遍历所有可能的选择
        for i in range(start, len(nums)):
            current_subset.append(nums[i])
            backtrack(i + 1, current_subset)  # 递归选择
            current_subset.pop()  # 回溯，撤销选择

    backtrack(0, [])
    return result

# 示例：
print(subsets([1, 2, 3]))  # 输出所有子集
```

```
try:
    while True:
        hefaxinyanzheng()
except EOFError:
    pass
```

例4：排列问题（回溯）

给定一个集合 `[1, 2, 3]`，输出所有排列。

python

```
def permute(nums):
    result = []

    def backtrack(current_permutation):
        # 终止条件：如果排列长度达到 nums 长度，保存当前排列
        if len(current_permutation) == len(nums):
            result.append(list(current_permutation))
            return

        # 递归选择：每次都尝试将未选择的元素加入排列
        for num in nums:
            if num not in current_permutation:
                current_permutation.append(num)
                backtrack(current_permutation)
                current_permutation.pop()  # 回溯，撤销选择

    backtrack([])
    return result
```

排

```
def combination_sum(candidates, target):
    result = []

    def backtrack(start, target, current_combination):
        # 如果剩余目标为 0，则找到了一个组合，保存当前组合
        if target == 0:
            result.append(list(current_combination))
            return
        # 遍历可选的数字，尝试将它们加入到当前组合中
        for i in range(start, len(candidates)):
            if candidates[i] > target:  # 如果当前数字大于剩余目标，剪枝
                continue
            current_combination.append(candidates[i])  # 做出选择
            backtrack(i, target - candidates[i], current_combination)  # 递归
            current_combination.pop()  # 撤销选择（回溯）

    backtrack(0, target, [])
    return result

# 示例：
candidates = [2, 3, 6, 7]
target = 7
print(combination_sum(candidates, target))  # 输出所有组合
```

排序自定义比较函数cmp_to_key()

```python
def hanoi(n, source, target, auxiliary):
    if n == 1:
        print(f"Move disk 1 from {source} to {target}")
        return
    # Step 1: Move n-1 disks from source to auxiliary
    hanoi(n - 1, source, auxiliary, target)
    # Step 2: Move the nth disk from source to target
    print(f"Move disk {n} from {source} to {target}")
    # Step 3: Move n-1 disks from auxiliary to target
    hanoi(n - 1, auxiliary, target, source)

# 示例:
n = 3  # 3 个圆盘
hanoi(n, 'A', 'C', 'B')  # A 为源柱子，C 为目标柱子，B 为辅助柱子
```

a.sort(key=cmp_to_key(compare))

bfs，dfs，juzhen，字符串，双指针，dp，递归，贪心

dfs（在DFS或者是BFS中传递参数要全，否则容易RTE）尤其是visit

本质是暴力枚举出所有的可能性

1.双十一购物
这一题主要就是找到不同种类商品的组成（运用到了DFS），然后维护最小价格。

```
#处理好输入输出
n,m=map(int,input().split())
price=[]
youhui=[]
dp=[0]*(m+1)
result=float("inf")
for i in range(n):
    price.append(input().split())
for i in range(m):
    youhui.append(input().split())
def DFS(i,sum1):
    global result
    if i==n:
        manjian=(sum1//300)*50
        jian=0
        for o in range(1,m+1):
            current=0
            for j in youhui[o-1]:
                 a,b=map(int,j.split("-"))
                 if dp[o]>=a:
                     current=max(current,b)
            jian+=current
        result=min(result,sum1-jian-manjian)
        return
    for i1 in price[i]:
        a,b=map(int,i1.split(":"))
        dp[a]+=b
        DFS(i+1,sum1+b)
        dp[a]-=b
DFS(0,0)
print(result)
```

2.两个孤岛之间的距离

先用DSF标记第一个孤岛，同时将第一个孤岛中的点都加入BFS的queue中。在用BFS找最短路径。

```
from collections import deque
dir=[(1,0),(-1,0),(0,-1),(0,1)]
q=deque()
def DFS(x,y,n,col,map,dir,q):
    map[x][y]=2
    q.append((x,y,0))
    for i in range(4):
        dx=dir[i][0]
        dy=dir[i][1]
        if 0<=x+dx<n and 0<=y+dy<col and map[x+dx][y+dy]==1:
            DFS(x+dx,y+dy,n,col,map,dir,q)
    return
def BFS(n,col,map,dir,q):
    while q:
        x,y,distance=q.popleft()
        for i in range(4):
            dx=dir[i][0]
            dy=dir[i][1]
            if 0<=x+dx<n and 0<=y+dy<col and map[x+dx][y+dy]==0:
                map[x+dx][y+dy]=2
                q.append((x+dx,y+dy,distance+1))
            if 0<=x+dx<n  and 0<=y+dy<col and map[x+dx][y+dy]==1:
                print(distance)
                exit(0)
def main():
    n = int(input())
    map = []
    for _ in range(n):
        current = [int(i) for i in input()]
        col = len(current)
        map.append(current)
    for i in range(n):
        Flag=False
        for j in range(col):
            if map[i][j]==1:
                DFS(i,j,n,col,map,dir,q)
                Flag=True
                break
        if Flag:
            break
    BFS(n,col,map,dir,q)
main()
```

3.受到祝福的平方
Math的用法
1.基本数学运算
math.sqrt（x）返回x的平方根
math.trunc(x) 返回x的整数部分
math.ceil(x) 向上取整
math.floor(x) 向下取整
math.gcd(x,y) 返回x和y的最大公约数
2.阶乘与组合、排列
math.comb(n,k)
从n个元素中选出k个元素进行组合

math.perm(n,k)
从n个元素中选取k个元素来进行排列
math.factorial(n)
返回n的阶乘
3.返回整数平方根
math.isqrt(17) 4
math.isqrt(16) 4

```
import math

def is_valid(i):
    if int(i) == 0:
        return False
    root = int(math.sqrt(i))
    if root * root != i:
        return False
    return True
Flag=False
def DFS(s):
    global Flag
    if len(s)==0:
        Flag=True
        print("Yes")
        exit(0)
    for i in range(1,len(s)+1):
        if is_valid(int(s[0:i])):
           DFS(s[i:len(s)])
    return
s=input()
DFS(s)
if not Flag:
    print("No")
```

4.走迷宫的多个版本

```
n,m=map(int,input().split())
maze=[]
count=0
for _ in range(n):
    a=list(map(int,input().split()))
    maze.append(a)
visited=[[False for _ in range(m)]for _ in range(n)]
directions=[(1,0),(-1,0),(0,1),(0,-1)]
def www(x,y):
    return 0<=x<n and 0<=y<m and maze[x][y]==0 and not visited[x][y]
def DFS(x,y):
    global count
    if x==n-1 and y==m-1:
        count+=1
        return
    visited[x][y]=True
    for i in range(4):
        dx= directions[i][0]
        dy= directions[i][1]
```

```
        if www(x+dx,y+dy):
            DFS(x+dx,y+dy)
    visited[x][y]=False
DFS(0,0)
print(count)

n, m = map(int, input().split())
maze = [[-1 for _ in range(m + 2)]]
for _ in range(n):
    a = list(map(int, input().split()))
    maze.append([-1]+a+[-1])
maze.append([-1 for _ in range(m + 2)])
maze[1][1] = "s"
maze[n][m] = "e"

count = 0
directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]
def migong(maze, row, col):
    global count
    for i in range(4):
        dx = directions[i][0]
        dy = directions[i][1]
        n_row = row + dx
        n_col = col + dy
        if maze[n_row][n_col] == "e":
            count += 1
            continue
        if maze[n_row][n_col] == 0:
            maze[row][col] = 1
            migong(maze, n_row, n_col)
            maze[row][col] = 0
    return

migong(maze,1,1)
print(count)

#中等走迷宫
#主要区别就是对步长有了限制要求，多传一个参数就好了
n, m,k= map(int, input().split())
maze = [[-1 for _ in range(m + 2)]]
for _ in range(n):
    a = list(map(int, input().split()))
    maze.append([-1]+a+[-1])
maze.append([-1 for _ in range(m + 2)])
maze[1][1] = "s"
maze[n][m] = "e"

directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]
result="NO"
def migong(maze, row, col,step):
    global result
    for i in range(4):
        dx = directions[i][0]
        dy = directions[i][1]
        n_row = row + dx
        n_col = col + dy
```

```
        if maze[n_row][n_col] == "e" and step==k-1:
            result="Yes"
        if maze[n_row][n_col] == 0:
            maze[row][col] = 1
            step+=1
            migong(maze, n_row, n_col,step)
            step-=1
            maze[row][col] = 0
    return
migong(maze,1,1,0)
print(result)


#权值走迷宫
directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]
n, m = map(int, input().split())
maze = []
visited = [[False for _ in range(m)] for _ in range(n)]
for i in range(n):
    a = list(map(int, input().split()))
    maze.append(a)
def is_valid(x, y):
    return 0 <= x < n and 0 <= y < m and not visited[x][y]
max_value = float('-inf')
def quanzhizoumigong(x, y, value):
    global max_value
    if x == n - 1 and y == m - 1:
        max_value = max(max_value, value+maze[x][y])
        return
    visited[x][y]=True
    for i in range(4):
        dx = directions[i][0]
        dy = directions[i][1]
        if is_valid(x + dx, y + dy):
            quanzhizoumigong(x+dx, y+dy, value+maze[x][y])
    visited[x][y]=False
    return
quanzhizoumigong(0, 0, 0)
print(max_value)
```

5.八皇后DFS

```
def match(test, dict1):
    left = test[0]
    right = test[1]
    comparison = test[2]
    correct_answer = sum([dict1[i] for i in left]) - sum([dict1[i] for i in
right])
    dict_answer = {"up": 1, "down": -1, "even": 0}
    return correct_answer == dict_answer[comparison]


def find_false_coin(test1, test2, test3):
    letters = ["A", "B", "C", "D", "E", "F", "G", "H", "I", "J", "K", "L"]
    dict1 = {m: 0 for m in letters}
```

```
    for coin in letters:
        dict1[coin] = -1
        if match(test1, dict1) and match(test2, dict1) and match(test3, dict1):
            print(f'{coin} is the counterfeit coin and it is light.')
            break
        dict1[coin] = 1
        if match(test1, dict1) and match(test2, dict1) and match(test3, dict1):
            print(f'{coin} is the counterfeit coin and it is heavy.')
            break
        dict1[coin] = 0


n = int(input())
for _ in range(n):
    test1 = input().split()
    test2 = input().split()
    test3 = input().split()
    find_false_coin(test1, test2, test3)
```

6.全排列（易错点：对列表进行的修改需要进行回退操作，）

```
n=int(input())
shuzu=[str(i) for i in range(1,n+1)]
list=[]
def dfs(current,a):
    if len(current)==n:
        list.append(current[:])
        return
    for i in a:
        if i not in current:
            current.append(i)
            dfs(current,a)
            current.pop()
dfs(current=[],a=shuzu)（要么全写成位置参数，要么全写成关键字参数）
for i in list:
    print(*i)
```

2.二分查找

1. 在已排序的数组中查找某个元素

最常见的二分查找应用场景是**在一个有序数组中查找某个元素**。假设数组是升序排列的，你可以通过二分查找快速定位到目标元素的位置（如果存在的话）。

例子：

- 给定一个有序数组 `[1, 2, 3, 4, 5, 6, 7, 8, 9]`，查找元素 5 是否在其中，并返回其下标。

python 复制代码

```
def binary_search(arr, target):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = left + (right - left) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1  # 如果找不到目标元素
```

```
l,n,m=map(int,input().split())
rock=[0]
for i in range(n):
    rock.append(int(input()))
rock.append(l)
def check(x):
    num=0
    now=0
    for i in range(1,n+2):
        if rock[i]-now<x:
            num+=1
        else:
            now=rock[i]
    if num>m:
        return True
    else:
        return False
#二分查找，上下边界分别设置为可能取到的最小值和最大值
lo,hi=0,l+1
ans=0
while lo<=hi:
    mid=(lo+hi)//2
    if check(mid):
        hi=mid-1
    else:
        ans=mid
        lo=mid+1
print(ans)
```

上述例题的情况就是想要找到一个符合check的要求且ans取得最大值，如果想要查找的不只是符合条件的某个数字，而且想要找到最优的一种情况，可以像以下例题中维护一个result保存值

2. 查找第一个或最后一个符合条件的元素

有时候，你需要查找数组中第一个符合某个条件的元素，或者查找最后一个符合条件的元素。这种问题可以使用变形的二分查找来解决。

1.1 查找第一个等于目标值的元素

- 如果目标元素有多个相同的值，且数组是排序的，那么可以通过二分查找找出第一个等于目标的元素。

python 复制代码

```python
def find_first(arr, target):
    left, right = 0, len(arr) - 1
    result = -1
    while left <= right:
        mid = left + (right - left) // 2
        if arr[mid] == target:
            result = mid
            right = mid - 1  # 继续查找左边的元素
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return result
```

4.贪心

等待问题类型的贪心算法：

一般将时间短的放在前面，时间长的放在后面

就可以直接使用 sort 正序排序，这样就是让每个人等待时间最短的方法

区间选点问题

最小化选点数的区间覆盖问题

按右端点排序，从最左端开始遍历。如果能覆盖就下一个，不能覆盖则将新的右端点作为选点，加入结果之中。

```python
def interval_selection(intervals):
    # 1. Sort intervals by the right endpoint
    intervals.sort(key=lambda x: x[1])

    points = []
    current_point = None

    # 2. Iterate through each interval
    for interval in intervals:
        # If the current interval is not covered by the current point
        if current_point is None or interval[0] > current_point:
            # Select the right endpoint of the current interval
            current_point = interval[1]
            points.append(current_point)

    return points
```

最大化不重叠区间数量

优先选择右端点最小的区间，因为选取右端点最小的区间能让我们有更多空间去选择后面的区间。按右端点升序排序，从左到右遍历区间，选择那些与前一个选定的区间不重叠的区间（即当前区间的左端点大于等于前一个区间的右端点）。

```python
def max_non_overlapping_intervals(intervals):
    # 1. Sort intervals by the right endpoint
    intervals.sort(key=lambda x: x[1])

    count = 0
    last_end = float('-inf')

    # 2. Iterate through each interval
    for interval in intervals:
        if interval[0] >= last_end:
            count += 1
            last_end = interval[1]

    return count
```

子：

```python

intervals = [(1, 3), (2, 5), (4, 7), (6, 8)]
print(max_non_overlapping_intervals(intervals))
```

课程安排（最小堆解决有限资源如何尽量少的资源满足尽量多的需求）

函数	功能描述	度
`heappush(heap, item)`	将元素 `item` 插入堆 `heap` 中，保持堆的性质。	O(log N)
`heappop(heap)`	弹出并返回堆顶元素（最小元素），并调整堆结构。	O(log N)
`heappushpop(heap, item)`	插入元素 `item` 并弹出堆顶元素（最小元素），合并操作效率较高。	O(log N)
`heapreplace(heap, item)`	弹出堆顶元素并插入元素 `item`，合并操作效率较高。	O(log N)

```python
import heapq  # 使用heapq模块来实现最小堆

def schedule_courses(courses):
    # 按照课程的开始时间进行排序
    courses.sort(key=lambda x: x[0])  # x[0]是课程的开始时间

    # 创建一个最小堆，堆元素表示教室的空闲时间
    min_heap = []

    for course in courses:
        start, end = course

        if min_heap and min_heap[0] <= start:
            # 如果堆顶的教室空闲时间早于等于当前课程的开始时间，则可以安排课程
            heapq.heappop(min_heap)  # 弹出最早空闲的教室
        # 无论是否有可用教室，都需要将当前课程的结束时间加入堆中
        heapq.heappush(min_heap, end)

    return len(min_heap)  # 堆中元素的数量即为所需的教室数量

# 示例：课程的时间为 [(start_time, end_time)]
courses = [(1, 3), (2, 5), (4, 7), (6, 8), (5, 6)]
print("需要的最小教室数量:", schedule_courses(courses))  # 输出 3
```