Python Cheat Paper

函数：

enumerate：给出数据与索引，得到二元组（编号，内容）

```python
def linear_search(arr, target):
    for i, element in enumerate(arr):
        if element == target:
            return i  # 返回目标元素的索引
    return -1  # 如果未找到目标元素，返回 -1

# 示例
arr = [3, 5, 2, 8, 1, 9, 4]
target = 8
result = linear_search(arr, target)
print(f"Target {target} found at index {result}")
# Target 8 found at index 3
```

```python
n = int(input()) #机器猫斗恶龙
dp = [0]*n #先初始化dp序列
levels = list(map(int,input().split()))
if levels[-1] >= 0: #结尾关卡不扣血
    dp[-1] = 1
else:
    dp[-1] = -levels[-1]+1
for i in range(n-2,-1,-1): #逆向检索
    dp[i] = max(dp[i+1]-levels[i],1)
print(dp[0])
```

count：列表和字符串都可以

dict：建立字典，可以先在 list 中装入 tuple，再用 for 循环，注意是每个字符（下）还是字符串映射

```python
for chars, num in mappings:
    for char in chars:
        char_to_num[char] = num
```

ord：查询 ascii 码

.pop：删去，括号里面放数据

Zip 可以实现矩阵行列转换，作用是把多个列表中的元素按照顺序对应并转化成元组，储存在新列表中。比如 zip([a,b,c],[1,2,3])得到新列表 [(a,1),(b,2),(c,3)]，也可以用 zip(*matrix)

```
python 复制代码

matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
```

正常 zip

```
python 复制代码

zip(matrix[0], matrix[1], matrix[2])
```

- 相当于把三行对应元素打包成列的元组
- 输出结果：

```
css 复制代码

[(1,4,7), (2,5,8), (3,6,9)]
```

每个元组就是原矩阵的一列

.index 可以得到列表中某个元素的序列

算法：

Greedy：双指针和二分法

经典 greedy 装包问题：

lambda 函数可以排序

技巧：是不是需要逆序检索？（右上）

```python
n = int(input())
lst = [0]*(n+1)
for i in range(1,n+1):
    for j in range(i*2,n+1,i):
        lst[j]+=i
```

对于分解质因数类的题，使用筛法算因数的和（右）

将列表中的数据组合成字符串：num = ''.join(num)

或者使用 print(*lst, sep=',')

建立矩阵可以使用 for 循环（右），遍历矩阵用两个 for：

```python
matrix = []
for i in range(5):
    row = list(map(int,input().split()))
    matrix.append(row)
```

字符串可以比较：'A'<'Z'<'a'<'z'

计数方法不止可以使用 count，还可以使用 for 循环加 get 建立词典（映射）：

```
counting = {}
for phone_num in number: #number是记录标准化后的电话号码的列表
    counting[phone_num] = counting.get(phone_num, 0) + 1
    #第一次运行counting没东西，输出默认值0+1=1，其后遇到相同数据则返回该键对应的值，如这个phone_num对应1
duplicates = [(num,count) for num,count in counting.items() if count > 1]
#.items返回所有键值对，最后把重复次数在duplicates列表中添加元组(num,count)
```

用 list.insert(value,index)插入数据

想转化为两位小数的形式可以使用 f'{(value):.2f}'

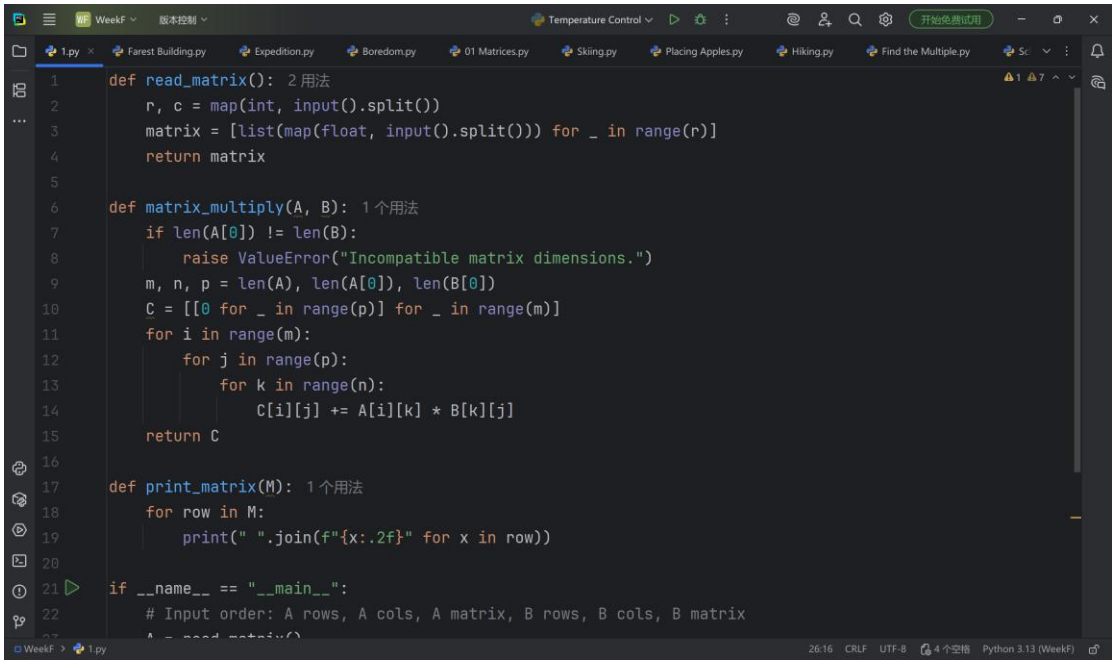
读取到某点终止可以使用 while True 下面循环中嵌套 if-break

贪心算法算图形题，常常把二维图形转化为线性的图像，即区间重叠问题，读取时从区间右端点排序开始计算（对于一个 tuple 我们可以采用 sort(key = lamdax：x[1])）比如在雷达安装问题中，对于平面内某点，可以安装雷达的点位只有在 x 轴上固定的一个区间内，我们可以比较几个区间的重叠

调用函数时多使用局部变量，运行主程序时这样写：if __name__ == '__main__':

函数名（）

矩阵乘法：



对于一个 queue，我们可以使用 deque 从左侧删除，降低复杂度也可以用第一行代码实现 queue 建立为空列表 dp 搜寻最长的递增列表：

最长递增子序列：

```python
nums = [10, 9, 2, 5, 3, 7, 101, 18]

n = len(nums)
dp = [1] * n

for i in range(n):
    for j in range(i):
        if nums[i] > nums[j]:
            dp[i] = max(dp[i], dp[j] + 1)

print(max(dp))
```

算日期:

```python
from datetime import date

# 定义两个日期
date1 = date(2025, 10, 27)  # 年, 月, 日
date2 = date(2025, 11, 5)

# 计算日期差
delta = date2 - date1

# 输出天数
print(delta.days)
```

进制转换（36 以下）:

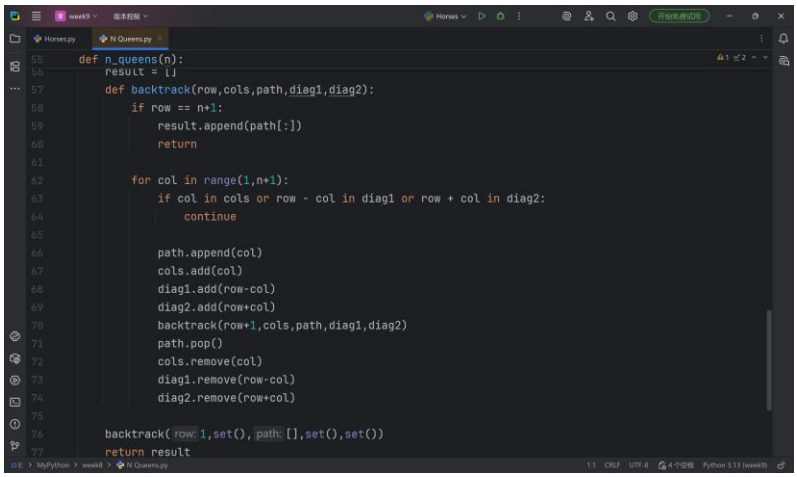
```python
binary_str = "101010"
decimal_num = int(binary_str, 2)   # base 2 → decimal
print(decimal_num)                 # 42

octal_str = "52"
decimal_num = int(octal_str, 8)    # base 8 → decimal
print(decimal_num)                 # 42

hex_str = "2a"
decimal_num = int(hex_str, 16)     # base 16 → decimal
print(decimal_num)                 # 42
```

```python
num = int(input())
print(bin(num)[2:])  # 二进制
print(oct(num)[2:])  # 八进制
print(hex(num)[2:])  # 十六进制
```

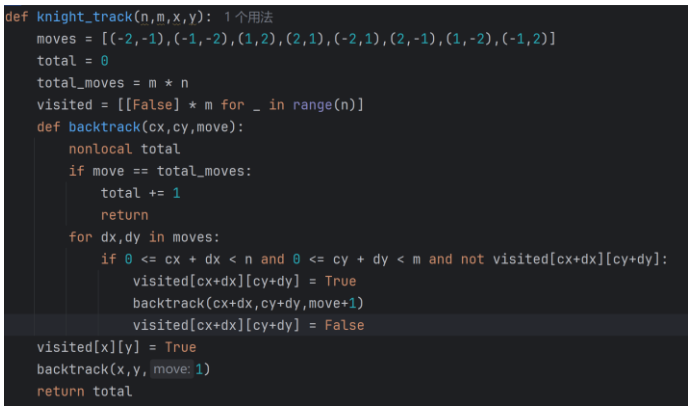
Ok-flag：防止出现循环检验（ok 放循环外）

回溯算法：用于处理排列组合等树状图问题，比如 n 皇后、全排列、子集和数独问题
先定标准（循环终止的标准），回溯到上一步再做其中图为 permutation，n_queens 和 Knight's Tour 的解答

```python
def backtrack(nums,path,result):  2 用法
    if len(nums) == 0:
        result.append(path[:])
        return

    for i in range(len(nums)):
        path.append(nums[i])
        remaining = nums[:i] + nums[i+1:]
        backtrack(remaining,path,result)
        path.pop()
```



```python
def n_queens(n):
    result = []
    def backtrack(row,cols,path,diag1,diag2):
        if row == n+1:
            result.append(path[:])
            return

        for col in range(1,n+1):
            if col in cols or row - col in diag1 or row + col in diag2:
                continue

            path.append(col)
            cols.add(col)
            diag1.add(row-col)
            diag2.add(row+col)
            backtrack(row+1,cols,path,diag1,diag2)
            path.pop()
            cols.remove(col)
            diag1.remove(row-col)
            diag2.remove(row+col)

    backtrack(row: 1,set(), path: [],set(),set())
    return result
```



```python
def knight_track(n,m,x,y):  1 个用法
    moves = [(-2,-1),(-1,-2),(1,2),(2,1),(-2,1),(2,-1),(1,-2),(-1,2)]
    total = 0
    total_moves = m * n
    visited = [[False] * m for _ in range(n)]
    def backtrack(cx,cy,move):
        nonlocal total
        if move == total_moves:
            total += 1
            return
        for dx,dy in moves:
            if 0 <= cx + dx < n and 0 <= cy + dy < m and not visited[cx+dx][cy+dy]:
                visited[cx+dx][cy+dy] = True
                backtrack(cx+dx,cy+dy,move+1)
                visited[cx+dx][cy+dy] = False
    visited[x][y] = True
    backtrack(x,y, move: 1)
    return total
```

递归程序在处理大规模问题时经常会遇到两个主要问题：递归深度限制 和 重复计算子问题。这两个问题可以通过以下两种方法来解决：增加递归深度限制：使用 sys.setrecursionlimit 来增加 Python 的递归深度限制。缓存中间结果：使用 functools.lru_cache 或其他形式的 memoization（记忆化）来避免重复计算。

加上下面两句更快（以 Fibonacci 数列为例（不过貌似 dp 更快））

```python
import sys
sys.setrecursionlimit(1 << 30)  # 将递归深度限制设置为 2^30
```

使用 `functools.lru_cache` 可以缓存函数的返回值，从而避免重复计算相同的子问题。这对于递归算法尤其有用，可以显著提高性能。

```python
from functools import lru_cache

@lru_cache(maxsize=None)
```

用 dfs 深度搜索，实现联通区域搜索，如晶矿和岛屿数量问题，下以晶矿为例：

```
def count_minerals():
    k = int(input())  # 读取测试数据组数
    for _ in range(k):  # 遍历每组测试数据
        n = int(input())  # 网格大小 n x n
        grid = []
        for _ in range(n):
            # 读取一行地图字符，并按字符分割为列表
            grid.append(list(input().strip()))

        # visited 用于标记某个位置是否访问过
        visited = [[False] * n for _ in range(n)]

        # 初始化两个晶矿统计计数
        red_count = 0
        black_count = 0

        # 深度优先搜索，搜索同一连通区域内的矿点
        def dfs(i, j, mineral_type):
            # 越界检查：若不在地图范围内，直接返回
            if i < 0 or i >= n or j < 0 or j >= n:
                return
            # 若当前格已访问过或不是同一种矿点，也不再继续
            if visited[i][j] or grid[i][j] != mineral_type:
                return
            # 标记当前格子已访问
            visited[i][j] = True

            # 向四个方向继续搜索
            dfs(i + 1, j, mineral_type)  # 下
            dfs(i - 1, j, mineral_type)  # 上
            dfs(i, j + 1, mineral_type)  # 右
            dfs(i, j - 1, mineral_type)  # 左

        # 遍历每一个位置
        for i in range(n):
            for j in range(n):
                # 只有未访问过并且为矿点（r/b）时才开始搜索
                if not visited[i][j] and grid[i][j] in ['r', 'b']:
                    mineral_type = grid[i][j]  # 获取矿类型
                    dfs(i, j, mineral_type)   # 搜索整个连通区域

                    # 搜索完说明这是一个完整晶矿，计数 +1
                    if mineral_type == 'r':
                        red_count += 1
                    else:
                        black_count += 1

        # 输出当前测试数据的结果
        print(f"{red_count} {black_count}")
```

实现全排列或者求子集的时候也可以采用这种方式避免重复计数，降低时间复杂度：

```
def backtracking(k,nums,path,result,start):  2 用法
    if len(path) == k:
        result.append(path[:])
        return
    for i in range(start,len(nums)):
        path.append(nums[i])
        backtracking(k,nums, path, result,i+1)
        path.pop()
    return result
n,k = map(int,input().split())
nums = list(map(int,input().split()))
ans = backtracking(k,nums, path: [], result: [], start: 0)
new_ans = []
for i in ans:
```

加入 start 之后可以保证每次都从已经搜寻过的数据的下一项开始。

并查集：

```python
n,m = map(int,input().split())
parent = list(range(n+1))
def find(x):  4 用法
    if parent[x] != x:
        parent[x] = find(parent[x])
    return parent[x]
def union(x,y):  1 个用法
    px = find(x)
    py = find(y)
    if px != py:
        parent[py] = px
for i in range(m):
    x,y = map(int,input().split())
    union(x,y)
classes = len(set(find(i) for i in range(1,n+1)))
print(classes)
```

获取不定行输入，忽略空格和空行（左），二维矩阵压缩成一维算最大子矩阵（右）：

```python
n = int(input())
m = []
while len(m) < n * n:
    line = input().strip()
    if not line:
        continue
    parts = line.split()
    m.extend(map(int,parts))
matrix = []
index = 0
for i in range(n):
    row = m[index:index+n]
    matrix.append(row)
    index += n
```

```python
max_sum = -float('inf')
for top in range(n):
    temp = [0] * n
    for bottom in range(top,n):
        for col in range(n):
            temp[col] += matrix[bottom][col]
        dp = [-127] * n
        dp[0] = temp[0]
        max_current = temp[0]
        for i in range(1,n):
            dp[i] = max(temp[i], dp[i-1] + temp[i])
            max_current = max(max_current, dp[i])
        max_sum = max(max_sum, max_current)
```

深搜（dfs）和广搜（bfs）：先考虑深搜的思路：

现有一个 n*m 大小的迷宫，其中`1`表示不可通过的墙壁，`0`表示平地。每次移动只能向上下左右移动一格（不允许移动到曾经经过的位置），且只能移动到平地上。求从迷宫左上角到右下角的所有可行路径的条数。

输入

第一行两个整数 n、m $(2 \le n \le 5, 2 \le m \le 5)$，分别表示迷宫的行数和列数；

接下来 n 行，每行 m 个整数（值为`0`或`1`），表示迷宫。

输出

一个整数，表示可行路径的条数。

对于这道题，在外面加一圈保护-1（与已经标记的墙做区分，保证回溯功能能正确执行），可以不需要判断语句，当碰到可以通的路的时候（即四个方向中的一个为 0），把现在在的位置封锁住，执行递归，并在递归执行结束后回溯通路。

如果需要实现计算到 end 需要多少步，可以在函数里面加上 step，最后统计 step+1 就行了。（右图）

```python
count = 0

def dfs(maze, x, y):  2 用法
    global count
    dx = [-1, 1, 0, 0]
    dy = [0, 0, -1, 1]
    for i in range(4):
        nx = x + dx[i]
        ny = y + dy[i]
        if nx == n and ny == m and matrix[nx][ny] == 0:
            count += 1
            continue
        if maze[nx][ny] == 0:
            maze[x][y] = 1
            dfs(maze, nx, ny)
            maze[x][y] = 0
    return count
```

```python
def dfs(maze, x, y, step):  2 用法
    global steps
    dx = [-1, 1, 0, 0]
    dy = [0, 0, -1, 1]
    for i in range(4):
        nx = x + dx[i]
        ny = y + dy[i]
        if nx == n and ny == m and matrix[nx][ny] == 0:
            steps.append(step + 1)
            continue
        if maze[nx][ny] == 0:
            maze[x][y] = 1
            dfs(maze, nx, ny, step+1)
            maze[x][y] = 0
dfs(matrix, x: 1, y: 1, step: 0)
if s in steps:
    print('Yes')
else:
    print('No')
```

如果希望实现路径可视化就是一个回溯的过程（就是一个 backtrack + dfs 综合问题），前部分的代码和思路完全一样，换皮罢了。

为什么不标记当前格子 (x, y)?

因为 path 在 DFS 调用之前已经决定:

- "下一步就是 (nx, ny)，那我应该先把它记下来"

👉 **本质原因:**

在回溯中，**路径是主角**，而不是 DFS 的探索过程。

你需要在每次递归进入下一层前:

1. 选择下一步
2. 记录该选择（append）
3. 占用该格（标记）
4. 深入递归
5. 回溯时撤销选择（pop + unmark）

整个过程是一条"选择 → 深入 → 回退"的递归树。

```python
    if nx == n and ny == m and matrix[nx][ny] == 0:
        path.append((nx, ny))
        paths.append(path[:])
        path.pop()
        continue

    if matrix[nx][ny] == 0:
        matrix[nx][ny] = 1          # 标记下一步为已走
        path.append((nx, ny))
        dfs(nx, ny, path)
        path.pop()                  # 回溯
        matrix[nx][ny] = 0          # 取消标记


# 从起点开始
matrix[1][1] = 1
dfs( x: 1,  y: 1,  path: [(1, 1)])
matrix[1][1] = 0
```

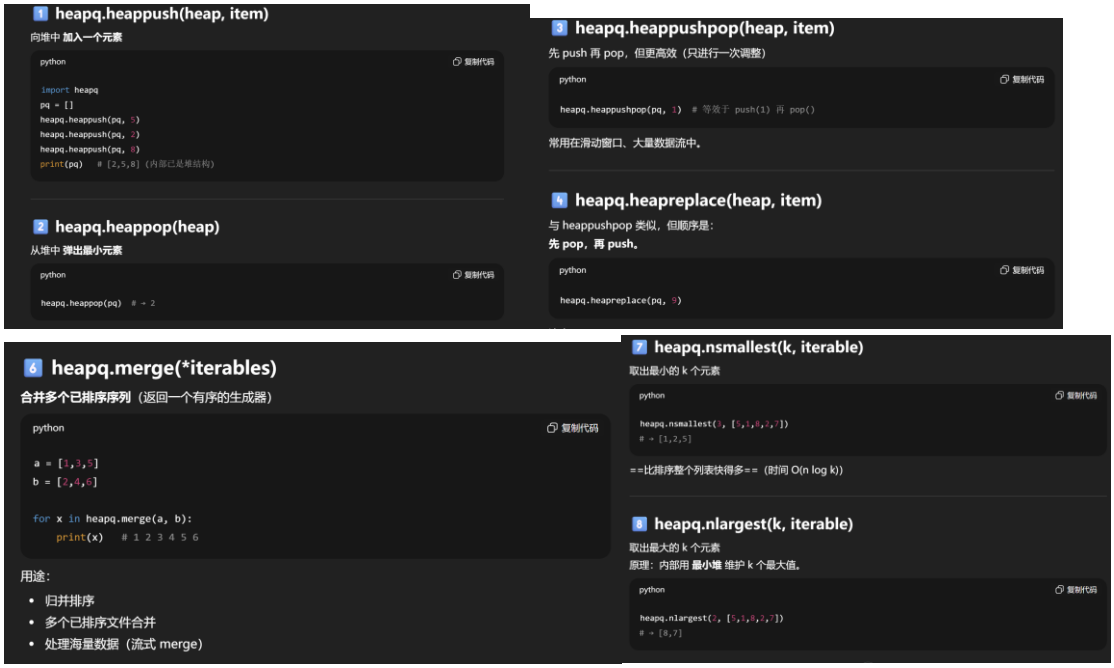
如果我希望计算不同路径的和呢？首先我考虑一个矩阵的路径和（这样不需要管有没有墙），然后我们再来看一个迷宫的路径和该怎么算。

```python
def dfs(x,y,total,count):  2 用法
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    for dx, dy in directions:
        nx, ny = x + dx, y + dy
        if 0 <= nx < n and 0 <= ny < m and not visited[nx][ny]:
            if nx == n-1 and ny == m-1:
                total.append(count + matrix[nx][ny])
                continue

            visited[nx][ny] = True
            dfs(nx,ny,total,count + matrix[nx][ny])
            visited[nx][ny] = False

    return total

visited[0][0] = True
ans = dfs( x: 0, y: 0, total: [],matrix[0][0])
print(max(ans))
```

可见，构造一个 visited 辅助矩阵判定是否经过是一个很不错的策略。不管采用这种列表里面套用 tuple 还是用两个列表直接给出 dx 和 dy，都需要把后续的内容嵌套在这个图寻之内。如果不搞一圈辅助-1 就需要判定是否越界（不然 RE：IndexError），判定到了终点就在 result 列表中添加这种可能的总和。走的时候标记下一步已走过再把当前 count 加上走的点即可。例如连通区域面积（个数只需要求 len（areas））

```python
def dfs(x,y):  2 用法
    visited[x][y] = True
    area = 1
    directions = [(1,0),(0,1),(-1,0),(0,-1)]
    for dx,dy in directions:
        nx = x + dx
        ny = y + dy
        if 0 <= nx < row and 0 <= ny < col and not visited[nx][ny] and matrix[nx][ny] == 1:
            area += dfs(nx, ny)
    return area

areas = []
for i in range(row):
    for j in range(col):
        if matrix[i][j] == 1 and not visited[i][j]:
            area = dfs(i,j)
            areas.append(area)

print(areas)
```

Heapq 和 Dijkstra 算法



下图为反向 dijkstra 算法，也就是说只是初始化值取相反数，加权改为减权，这样得到的就是最大结果的相反数（还是最小结果），return -result 就好了

```python
def dijkstra(maze,value,r,c):  1 个用法
    starter = float('inf')
    current = [[starter] * c for _ in range(r)]
    current[0][0] = -value[0][0]
    pq = [(current[0][0],0,0)]
    while pq:
        result,x,y = heapq.heappop(pq)
        if result > current[x][y]:
            continue
        if x == r-1 and y == c-1:
            return -result
        for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
            nx = x + dx
            ny = y + dy
            if 0 <= nx < r and 0 <= ny < c and maze[nx][ny] == 0:
                new_result = result - value[nx][ny]
                if new_result < current[nx][ny]:
                    current[nx][ny] = new_result
                    heapq.heappush(pq, item: (new_result,nx,ny))
    return -1
```

判定完全平方:

```python
import math
def is_square(x):  1 个用法
    if x <= 0:
        return False
    r = int(math.isqrt(x))
    return r * r == x
```

BFS 先双端序列，再依次枚举（当然可以提前判定 break 用于减少时间），其模板如下：对于切片问题，也可以采用 BFS，如下题。BFS 用于处理一些最短路径问题（有权值就用 Dijkstra）

广度优先搜索 (BFS)一般由队列实现,且总是按层次的顺序进行遍历，其基本写法如下(可作BFS模板用):

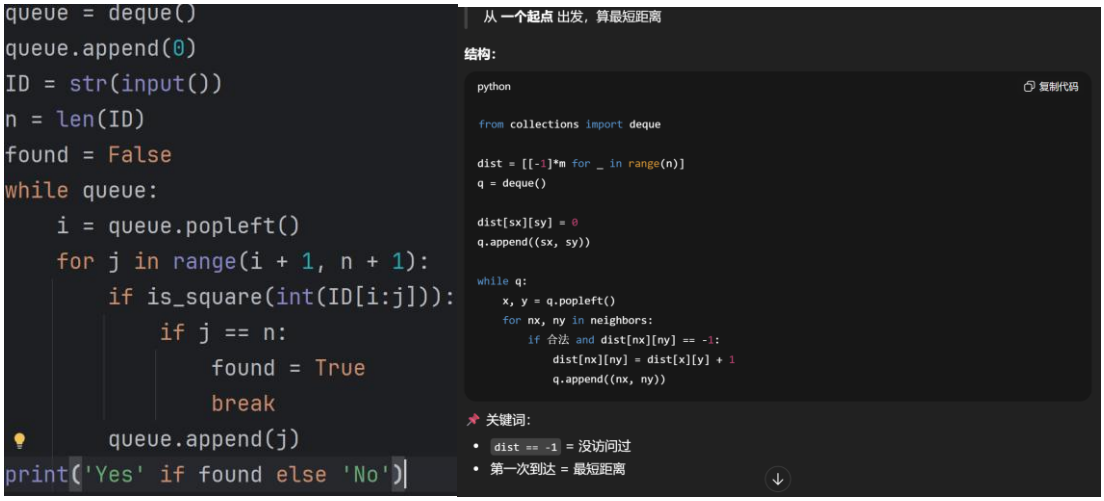
我们使用from collections import deque就满足要求，适用于需要频繁从队列的两端进行操作的场景，如广度优先搜索（BFS）、滑动窗口等问题。

> from queue import Queue适用于多线程编程中，需要在多个线程之间安全地共享和传递数据的场景。提供线程安全的特性，内置锁机制，可以在多线程环境中安全地使用。支持阻塞操作，如 `get` 和 `put` 方法可以设置超时时间，等待队列中有数据可用或空间可用。不支持从队列两端进行操作，只能从一端进行插入和删除。

```python
from collections import deque

def bfs(start, end):
    q = deque([(0, start)])  # (step, start)
    in_queue = {start}


    while q:
        step, front = q.popleft() # 取出队首元素
        if front == end:
            return step # 返回需要的结果，如：步长、路径等信息

        # 将 front 的下一层结点中未曾入队的结点全部入队q，并加入集合in_queue设置为已入队
```

下面是对该模板中每一个步骤的说明,请结合代码一起看:

① 定义队列 q，并将起点(0, start)入队，0表示步长目前是0。
② 写一个 while 循环，循环条件是队列q非空。
③ 在 while 循环中，先取出队首元素 front。
④ 将front 的下一层结点中所有**未曾入队**的结点入队，并标记它们的层号为 step 的层号加1，并加入集合in_queue设置为已入队。
⑤ 返回 ② 继续循环。



Dijkstra 适用于权值不同的 bfs（即走每一步消耗不一样），使用一定要初始化一个

```python
def dijkstra(x1,y1,x2,y2):  1 个用法
    if matrix[x1][y1] == '#' or matrix[x2][y2] == '#':
        return 'NO'
    dist = [[float("inf")] * n for _ in range(m)]
    dist[x1][y1] = 0
    pq = []
    heapq.heappush(pq, item: (0,x1,y1))
    directions = [(1,0),(0,1),(-1,0),(0,-1)]
    while pq:
        cost, x, y = heapq.heappop(pq)
        if cost > dist[x][y]:
            continue
        if x == x2 and y == y2:
            return cost
        for dx,dy in directions:
            nx,ny = x+dx,y+dy
            if 0 <= nx < m and 0 <= ny < n and matrix[nx][ny] != '#':
                h = abs(int(matrix[nx][ny])-int(matrix[x][y]))
                if dist[nx][ny] > cost + h:
                    dist[nx][ny] = cost + h
                    heapq.heappush(pq, item: (dist[nx][ny],nx,ny))
    return 'NO'
```

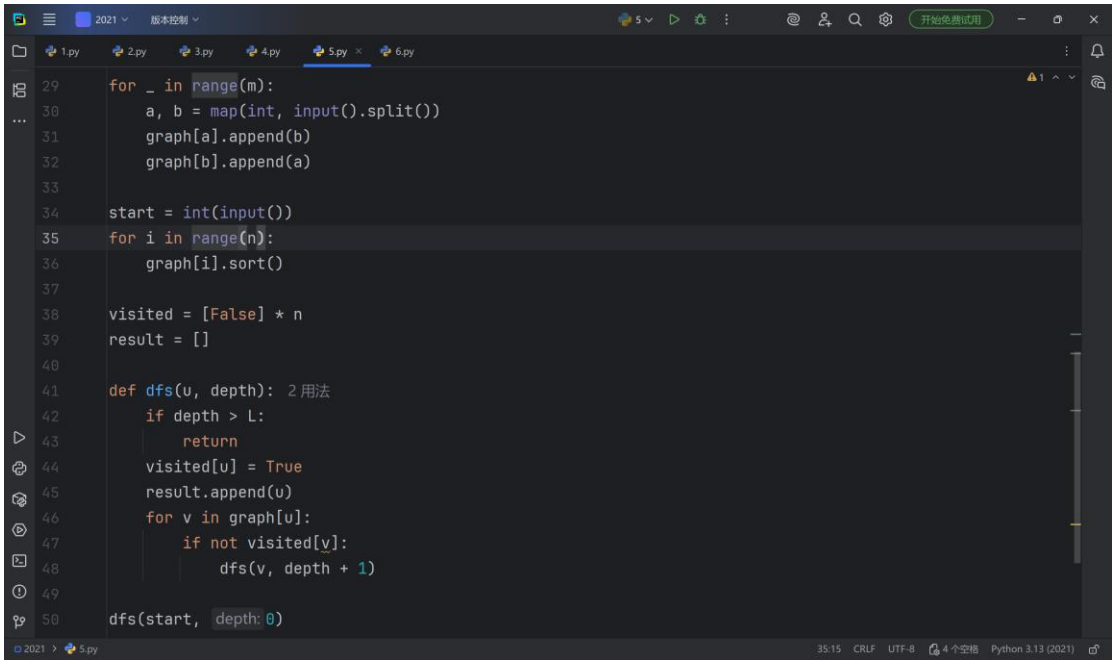
质数表:

```python
def build_prime_table(n):
    # 初始化质数表，is_prime[i] 为 True 表示 i 是质数
    is_prime = [True] * (n + 1)
    is_prime[0] = is_prime[1] = False

    i = 2
    while i * i <= n:
        if is_prime[i]:
            # 标记 i 的倍数为非质数
            for j in range(i * i, n + 1, i):
                is_prime[j] = False
        i += 1

    # 提取所有质数
    primes = [i for i in range(2, n + 1) if is_prime[i]]
    return primes, is_prime
```

最长回文子序列:

```python
def longestPalindrome(s):  1 个用法
    n = len(s)
    if n < 2:
        return ''.join(s)
    else:
        dp = [[0] * n for _ in range(n)]
        for i in range(n):
            dp[i][i] = 1
        for length in range(2, n+1):
            for i in range(n-length+1):
                j = i + length - 1
                if s[i] == s[j]:
                    dp[i][j] = dp[i+1][j-1] + 2 if length > 2 else 2
                else:
                    dp[i][j] = max(dp[i+1][j], dp[i][j-1])
                    result = []
                    i, j = 0, n - 1
                    while i <= j:
                        if s[i] == s[j]:
                            result.append(s[i])
                            i += 1
                            j -= 1
```

DLS（即有深度限制的 DFS）：



对于任意一个 DFS 可以用上面的方法把一条边记录下来。

如果我希望实现 BFS 的最短路同时实现路径回溯呢?

A:考虑构建 prev 矩阵，记录来时路（最后一定 reverse）

```python
dirs = [(1, 0), (-1, 0), (0, 1), (0, -1)]

# prev[x][y] = (px, py)
prev = [[None] * m for _ in range(n)]

# BFS 队列
q = deque()
q.append((0, 0))
prev[0][0] = (-1, -1)  # 起点标记

found = False
while q:
    x, y = q.popleft()
    if (x, y) == (n - 1, m - 1):
        found = True
        break
    for dx, dy in dirs:
        nx, ny = x + dx, y + dy
        if 0 <= nx < n and 0 <= ny < m:
            if maze[nx][ny] == 0 and prev[nx][ny] is None:
                prev[nx][ny] = (x, y)
                q.append((nx, ny))
```

```python
# 如果没找到终点
if not found:
    print(-1)
    exit()

# 反向还原路径
path = []
cur = (n - 1, m - 1)
while cur != (-1, -1):
    path.append(cur)
    cur = prev[cur[0]][cur[1]]

path.reverse()
```

BFS（求最短步长）如果需要路径可视化，可以采用在 queue 里面加入次序

```python
from collections import deque
def bfs(n,m):  1 个用法
    queue = deque()
    queue.append((0,n,''))
    inq = set()
    while queue:
        step,i,path = queue.popleft()
        if i == m:
            return (step,path)
        if 3*i not in inq:
            inq.add(3*i)
            queue.append((step+1,3*i,path + 'H'))

        if i // 2 not in inq:
            inq.add(i//2)
            queue.append((step+1,i//2,path + '0'))
```

经典习题:

1.Cube

```
visited = [False] * 4
def dfs(word,pos):  2 用法
    if len(word) > 4:
        return False

    if pos == len(word):
        return True

    for i in range(4):
        if not visited[i] and word[pos] in cube[i]:
            visited[i] = True
            if dfs(word,pos+1):
                return True
            visited[i] = False

    return False
```

2.洋葱

```
for i in range(n):
    row = list(map(int, input().split()))
    matrix.append(row)
t = n // 2
lst = [0] * t
result = [0] * t
if n % 2 == 1:
    for i in range(t):
        total = 0
        data = list(range(-i, i + 1))
        for j in data:
            for k in data:
                total += matrix[t-1+j][t-1+k]
        lst[i] = total
        result[i] = total

    for i in range(t-1):
        lst[i+1] = result[i+1] - result[i]
```

3.约瑟夫问题:

```
while True:
    n,p,m = map(int,input().split())
    if n==0 and m==0 and p==0:
        break
    numbers = list(range(1,n+1))
    index = p - 1
    ans = []
    while len(numbers) > 1:
        index  = (index + m - 1) % len(numbers)
        ans.append(numbers.pop(index))
    ans.append(numbers.pop(0))
    print(*ans,sep=',')
```

4.假期生活:

```
for _ in range(n):
    s, e, v = input().split()
    s = date_to_int(s)
    e = date_to_int(e)
    v = int(v)
    if e <= 51:
        plans.append((s, e, v))
if not plans:
    print(0)
    exit()
plans.sort(key=lambda x: x[1])
m = len(plans)
dp = [0] * m
for i in range(m):
    s_i, e_i, v_i = plans[i]
    best = v_i
    for j in range(i - 1, -1, -1):
        if plans[j][1] < s_i:
            best = dp[j] + v_i
            break
    dp[i] = max(dp[i - 1] if i > 0 else 0, best)
```

5.蛇入迷宫：

```
from collections import deque
n = int(input())
maze = []
visited = [[[False]*2 for _ in range(n)] for _ in range(n)]
visited[0][0][0] = True
for i in range(n):
    maze.append(list(map(int, input().split())))
queue = deque()
queue.append((0,0,0,0))
# 我们定义0为横着，1为竖着，数组的顺序是row，col，pos
# [False]*2初始每个元素都是一个列表（两个false表示方向没有经历过）
while queue:
    pos,step,tail_x,tail_y = queue.popleft()
    if tail_x == n-1 and tail_y == n-2 and pos == 0:
        print(step)
        break
```

```
if pos == 0:
    if maze[tail_x][tail_y] == 1 or maze[tail_x][tail_y + 1] == 1:
        continue
    if tail_y + 2 < n and maze[tail_x][tail_y + 2] == 0:
        if not visited[tail_x][tail_y + 1][0]:
            visited[tail_x][tail_y + 1][0] = True
            queue.append((0, step + 1, tail_x, tail_y + 1))

    if tail_x + 1 < n and maze[tail_x + 1][tail_y] == 0 and maze[tail_x + 1][tail_y + 1] == 0:
        if not visited[tail_x + 1][tail_y][0]:
            visited[tail_x + 1][tail_y][0] = True
            queue.append((0, step + 1, tail_x + 1, tail_y))

    if 0 <= tail_x < n-1 and maze[tail_x+1][tail_y+1] == 0 and maze[tail_x+1][tail_y] == 0:
        if not visited[tail_x][tail_y][1]:
            visited[tail_x][tail_y][1] = True
            queue.append((1,step+1,tail_x,tail_y))
```

Pos==1 的情况完全一样、

6.登山

```
def dijkstra(x1,y1,x2,y2):  1 个用法
    if matrix[x1][y1] == '#' or matrix[x2][y2] == '#':
        return 'NO'
    dist = [[float("inf")] * n for _ in range(m)]
    dist[x1][y1] = 0
    pq = []
    heapq.heappush(pq, item: (0,x1,y1))
    directions = [(1,0),(0,1),(-1,0),(0,-1)]
    while pq:
        cost, x, y = heapq.heappop(pq)
        if cost > dist[x][y]:
            continue
        if x == x2 and y == y2:
            return cost
        for dx,dy in directions:
            nx,ny = x+dx,y+dy
            if 0 <= nx < m and 0 <= ny < n and matrix[nx][ny] != '#':
                h = abs(int(matrix[nx][ny])-int(matrix[x][y]))
                if dist[nx][ny] > cost + h:
                    dist[nx][ny] = cost + h
                    heapq.heappush(pq, item: (dist[nx][ny],nx,ny))
```

7.滑雪

```
r,c = map(int, input().split())
matrix = []
for i in range(r):
    matrix.append(list(map(int, input().split())))
dp = [[1] * c for _ in range(r)]
cells = []
directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
for i in range(r):
    for j in range(c):
        cells.append((matrix[i][j], i, j))
        cells.sort()
for h,i,j in cells:
    for dx,dy in directions:
        nx, ny = i + dx, j + dy
        if 0 <= nx < r and 0 <= ny < c and matrix[nx][ny] > h:
            dp[nx][ny] = max(dp[i][j] + 1, dp[nx][ny])

local_max = []
for i in range(r):
    local_max.append(max(dp[i]))
print(max(local_max))
```

8. 0，1 矩阵

```
from collections import deque
row,col = map(int, input().split())
matrix = []
for i in range(row):
    matrix.append(list(map(int, input().split())))
dist = [[-1] * col for _ in range(row)]
queue = deque()
for i in range(row):
    for j in range(col):
        if matrix[i][j] == 0:
            queue.append((i, j))
            dist[i][j] = 0
directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
while queue:
    x, y = queue.popleft()
    for dx, dy in directions:
        nx, ny = x + dx, y + dy
        if 0 <= nx < row and 0 <= ny < col and dist[nx][ny] == -1:
            dist[nx][ny] = dist[x][y] + 1
            queue.append((nx, ny))
```

9.打家劫舍

```
class Solution:
    def rob(self, nums: list[int]) -> int:
        n = len(nums)
        if n == 0:
            return 0
        if n == 1:
            return nums[0]

        dp = [0] * n
        dp[0] = nums[0]
        dp[1] = max(nums[0], nums[1])

        for i in range(2, n):
            dp[i] = max(dp[i - 1], dp[i - 2] + nums[i])

        return dp[n - 1]
```

10.Flowers

```
for _ in range(t):
    a,b = map(int, input().split())
    lst.append([a,b])
max_b = max(b for a, b in lst)
dp = [0] * (max_b+1)
dp[0] = 1
if k == 1:
    dp[1] = 2
else:
    dp[1] = 1
for i in range(2, max_b+1):
    if i >= k:
        dp[i] = (dp[i-1] + dp[i - k]) % mod
    else:
        dp[i] = dp[i-1]
pref = [0] * (max_b + 2)
for i in range(1, max_b + 1):
    pref[i] = (pref[i-1] + dp[i]) % mod
for a, b in lst:
    print((pref[b] - pref[a-1]) % mod)
```

11.采药

```
T,M = map(int, input().split())
times = []
values = []
dp = [[0] * (T + 1) for _ in range(M + 1)]
for _ in range(M):
    t,v = map(int, input().split())
    times.append(t)
    values.append(v)
for i in range(1,M+1):
    for j in range(T + 1):
        dp[i][j] = dp[i - 1][j]
        if j >= times[i-1]:
            dp[i][j] = max(dp[i][j], dp[i - 1][j - times[i-1]] + values[i - 1])
print(dp[M][T])
```