保留两位小数："%.2f"%n

列表：

1. 将字符串转化成列表：eg:li=[] ， li.extend('abc') -> li=['a','b','c']
   列表转化成字符串：''.join(l)

2.del & remove

del: del li ， del li[3] ， del li[1:3]

remove：li=[1,1,2,3]， li.remove(1) -> li=[1,2,3](注意：只会删除第一个出现的元素！)

3.pop： li.pop(),删除列表的最后一个元素

li.pop(k):删除指定下标的元素(根据元素下标进行删除)

li=[1,2,3] li.pop(2) -> li=[1,2]

4.index ： name.index(查找元素，起始值，终止值（但不包括这个值自己，类似于 range））
eg: li=[1,2,3,1,1]， print(li.index(1,0,3))-> 0 (注意：若重复元素，index 只能输出第一个元素的索引)

5.count ： li=[1,2,3,1,1]， print(li.count(1)) -> 3

排序：

1.a.sort()&sorted(a)

2.倒序 a.sort(reverse=True)

3. eg1:sorted_list = sorted(a, key = lambda x: x[0]):意为按照子列表的第一个元素对所有子列表进行排序

   eg2:sorted_list = sorted(a, key = lambda x: （x[0],x[1]）):意为按照子列表第一个元素进行排序，若第一个元素相同，则再按照第二个元素大小进行排序

   eg3:sorted_list = sorted(a, key = lambda x: x[0],reverse=True):意为按照子列表第一个元素从大到小逆序对整个子列表进行排序

   eg4:sorted_list = sorted(a, key = lambda x: （x[0],-x[1]）):意为按照子列表第一个元素进行排序，若第一个元素相同，则再按照第二个元素大小的逆序进行排序

4.双列表：li=[1,2,3,6,4] nums=[0,3,1,7,4] li_=sorted(li,key=lambda x: nums[li.index(x)])
print(li_) li-=[1,3,2,4,6](意为通过 li 中的下标对应的 nums 的大小排序并返回 li 的排序)
其他用法：eg: li_sorted=sorted(li,key=lambda x:(x[0]*x[1]))

保护圈模板： 1.grid = [' ' * (w + 2)] + [' ' + input() + ' ' for _ in range(h)] + [' ' * (w + 2)]（字符串写法）

字典： 花括号，冒号，注意引号内是否有多余的空格

```
dic= {'zero':0, 'one':1, 'two':2, 'three':3, 'four':4,'five':5,
      'six':6, 'seven':7, 'eight':8, 'nine':9, 'ten':10,
      'eleven':11, 'twelve':12, 'thirteen':13, 'fourteen':14, 'fifteen':15,
      'sixteen':16, 'seventeen':17, 'eighteen':18, 'nineteen':19, 'twenty':20,
      'thirty':30, 'forty':40, 'fifty':50, 'sixty':60, 'seventy':70, 'eighty':80, 'ninety':90,
      'hundred':100, 'thousand':1000, 'million':1000000}
```

字符串： 想在输入的时候忽略符号:

```
n=int(input())
Haab_dates=[]
for i in range(n):
    input_str=input()
    day,month,year=input_str.replace( __old: '.', __new: ' ').split()
    Haab_dates.append((int(day),Haab_months.index(month),int(year)))
```

dp 数组：

小偷背包：

```
n,b=map(int, input().split())
price=[0]+[int(i) for i in input().split()]
weight=[0]+[int(i) for i in input().split()]
bag=[[0]*(b+1) for _ in range(n+1)]
for i in range(1,n+1):
    for j in range(1,b+1):
        if weight[i]<=j:
            bag[i][j]=max(price[i]+bag[i-1][j-weight[i]], bag[i-1][j])
        else:
            bag[i][j]=bag[i-1][j]
print(bag[-1][-1])
```

田忌赛马：

```
while True:
    n=int(input())
    if n==0:
        break
    a=list(map(int,input().split()))
    b=list(map(int,input().split()))
    a.sort(reverse=True)
    b.sort(reverse=True)
    dp=[[0]*(n+1) for _ in range(n+1)]#田用前i匹，王用前j匹，田赢的最大次数
    #记赢一局+2，输一局0，平局+1
    for i in range(1,n+1):
        for j in range(1,n+1):
            if a[i-1]>b[j-1]:#比较田第i匹和王第j匹
                dp[i][j]=max(dp[i-1][j],dp[i][j-1],dp[i-1][j-1]+2)
            if a[i-1]==b[j-1]:
                dp[i][j]=max(dp[i-1][j],dp[i][j-1],dp[i-1][j-1]+1)
            if a[i-1]<b[j-1]:
                dp[i][j]=max(dp[i-1][j],dp[i][j-1],dp[i-1][j-1])
    print(200*(dp[n][n]-n))
```

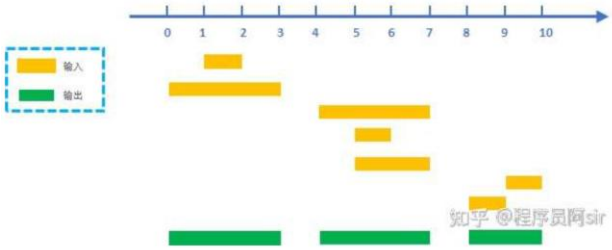
最长非递增子序列：

```
def max_intercepted_missiles(k, heights):
    # Initialize the dp array
    dp = [1] * k

    # Fill the dp array
    for i in range(1, k):
        for j in range(i):
            if heights[i] <= heights[j]:
                dp[i] = max(dp[i], dp[j] + 1)

    # The result is the maximum value in dp array
    return max(dp)
```

区间问题：

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）。最后问合并之后的**区间。**



区间合并问题示例：合并结果包含3个区间
【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。从前往后枚举每一个区间，判断是否应该将当前区间视为新区间。

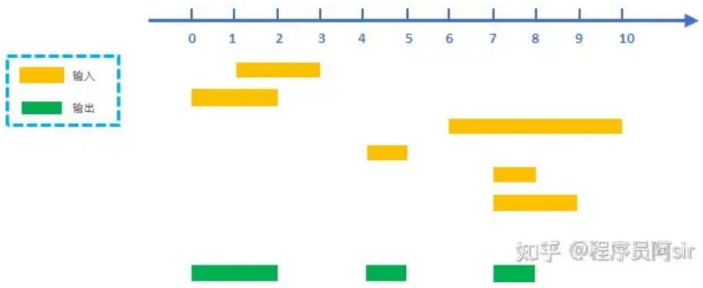
假设当前遍历到的区间为第i个区间 [l_i，r_i]，有以下两种情况：

- 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）。
- 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：

给出一堆区间，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。



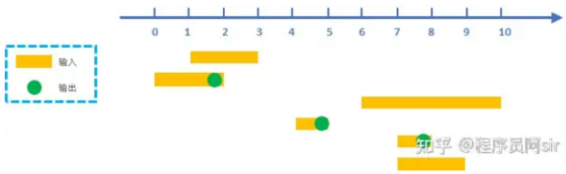
选择不相交区间问题示例：结果包含3个区间
【步骤一】：按照区间**右端点**从小到大排序。

【步骤二】：从前往后依次枚举每个区间。

```
list.sort(key=lambda x:x[1])
ed=list[0][1]
ans=[list[0]]
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）



区间选点问题示例，最终至少选择3个点
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：

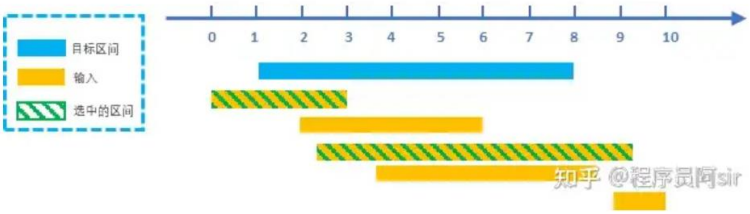
- l_i <=ed：说明当前区间与前面区间有交集，前面已经选点了。因此直接跳过。
- l_i > ed：说明当前区间与前面没有交集。因此选中当前区间，并设置 ed = r_i。

```
list.sort(key=lambda x:x[1])
ed=list[0][1]
ans=[list[0][1]]
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。

如下图所示：



区间覆盖问题示例，最终至少选择2个区间才能覆盖目标区间
【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。

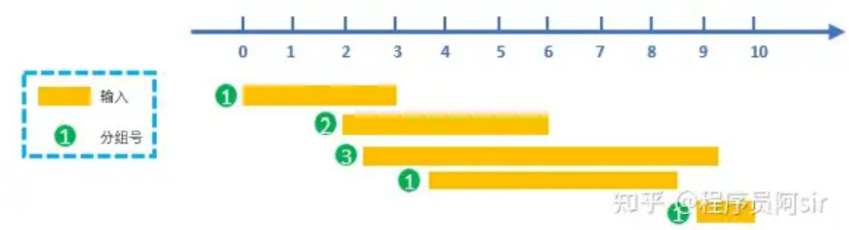
假设右端点最大的区间是第i个区间，右端点为 r_i。

最后将目标区间的start更新成r_i

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

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。



区间分组问题示例，最少分成3个组

【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)
```

BFS：

BFS 通常用来处理最短路问题(连通分支、走迷宫等问题一般用 DFS 解决;对于最短路问题，由于用 DFS 可能需要遍历所有可能路径，BFS 的时间复杂度常常会小得多)

寻宝问题：(1 是目标点，2 是陷阱不能经过)

其实所有求最短、最长的问题都能用heapq实现，在图搜索中搭配bfs尤其好用。

```
#23 工学院 苏王捷
import heapq
def bfs(x,y):
    d=[[-1,0],[1,0],[0,1],[0,-1]]
    queue=[]
    heapq.heappush(queue,[0,x,y])
    check=set()
    check.add((x,y))
    while queue:
        step,x,y=map(int,heapq.heappop(queue))
        if martix[x][y]==1:
            return step
        for i in range(4):
            dx,dy=x+d[i][0],y+d[i][1]
            if martix[dx][dy]!=2 and (dx,dy) not in check:
                heapq.heappush(queue,[step+1,dx,dy])
                check.add((dx,dy))
    return "NO"

m,n=map(int,input().split())
martix=[[2]*(n+2)]+[[2]+list(map(int,input().split()))+[2] for i in range(m)]+[[2]*(n+2)]
print(bfs(1,1))
```

DFS：

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 = [...]
...
visited = [[False]*n for _ in range(m)]
def dfs(x,y):
    global area
    if vis[x][y]:
        return
    visited[x][y] = True
    ...
    for dx,dy in directions:
        nx,ny=x+dx,y+dy
        if 0<=nx<m and 0<=ny<n and vis[nx][ny] and ...:
            dfs(nx,ny)
#此处还可以在dfs前不用标记vis，在for循环里：
#vis[nx][ny]=1
#dfs(nx,ny)
#vis[nx][ny]=0
#这样自身形成回溯
for i in range(m):
    for j in range(n):
        if not visited[i][j]:
            dfs(i,j)
```

矩阵最大权值路径：需要导出路径的

```
def dfs(x,y,now_value):
    global max_value,opt_path
    # 如果到达右下角，更新最大权值和最优路径
    if x == n - 1 and y == m - 1:
        if now_value > max_value:
            max_value = now_value
            opt_path = temp_path[:]
        return

    # 标记当前位置为已访问
    visited[x][y] = True

    # 尝试向四个方向移动
    for dx, dy in directions:
        next_x, next_y = x + dx, y + dy
        if 0 <= next_x < n and 0 <= next_y < m and not visited[next_x][next_y]:
            next_value = now_value + maze[next_x][next_y]
            temp_path.append((next_x, next_y))
            dfs(next_x, next_y, next_value)
            temp_path.pop()  # 回溯
    # 取消当前位置的访问标记
    visited[x][y] = False

# 读取输入
n, m = map(int, input().split())
maze = [list(map(int, input().split())) for _ in range(n)]

# 初始化变量
max_value = float('-inf')
opt_path = []
temp_path = [(0, 0)]
visited = [[False] * m for _ in range(n)]
directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]

# 从左上角开始DFS搜索
dfs( x: 0,  y: 0, maze[0][0])

# 输出最优路径
for x, y in opt_path:
    print(x + 1, y + 1)
```

马走日：

```
maxn = 10;
sx = [-2,-1,1,2, 2, 1,-1,-2]
sy = [ 1, 2,2,1,-1,-2,-2,-1]

ans = 0;

def Dfs(dep: int, x: int, y: int):
    #是否已经全部走完
    if n*m == dep:
        global ans
        ans += 1
        return

    #对于每个可以走的点
    for r in range(8):
        s = x + sx[r]
        t = y + sy[r]
        if chess[s][t]==False and 0<=s<n and 0<=t<m :
            chess[s][t]=True
            Dfs(dep+1, s, t)
            chess[s][t] = False; #回溯


for _ in range(int(input())):
    n,m,x,y = map(int, input().split())
    chess = [[False]*maxn for _ in range(maxn)]  #False表示没有走过
    ans = 0
    chess[x][y] = True
    Dfs(1, x, y)
    print(ans)
```

八皇后：

```
def is_safe(board, row, col):
    # 检查当前位置是否安全
    # 检查同一列是否有皇后
    for i in range(row):
        if board[i][col] == 1:
            return False
    # 检查左上方是否有皇后
    i = row - 1
    j = col - 1
    while i >= 0 and j >= 0:
        if board[i][j] == 1:
            return False
        i -= 1
        j -= 1
    # 检查右上方是否有皇后
    i = row - 1
    j = col + 1
    while i >= 0 and j < 8:
        if board[i][j] == 1:
            return False
        i -= 1
        j += 1
    return True
```

```
def solve_n_queens(board, row, solutions):
    # 递归回溯求解八皇后问题
    if row == 8:
        # 找到一个解，将解添加到结果列表
        solutions.append([board[i].copy() for i in range(8)])
        return
    for col in range(8):
        if is_safe(board, row, col):
            # 当前位置安全，放置皇后
            board[row][col] = 1
            # 继续递归放置下一行的皇后
            solve_n_queens(board, row + 1, solutions)
            # 回溯，撤销当前位置的皇后
            board[row][col] = 0

# 初始化棋盘
board = [[0] * 8 for _ in range(8)]
solutions = []
# 求解八皇后问题
solve_n_queens(board, 0, solutions)
# 输出结果
```