

目录

1. 字符串，集合，列表
2. 字典
3. math, 缓存函数结果，素数，递归深度，保护圈
4. 日期，collections
5. dfs
6. 二分查找
7. 排序
8. 背包
9. 背包
10. 背包
11. 区间DP
12. 状压DP
13. 优先队列，堆
14. 工具
15. 区间
16. 区间
17. 区间
18. END

字符串

- `str(object)`: 将对象转换为字符串。
- `repr(object)`: 获取对象的可打印表示形式 (通常用于调试)。

常用方法

- `string.upper()`: 返回字符串的大写版本。
- `string.lower()`: 返回字符串的小写版本。
- `string.capitalize()`: 返回首字母大写的字符串。
- `string.title()`: 返回每个单词首字母大写的字符串。
- `string.swapcase()`: 大小写互换。

查找和替换

- `string.find(sub[, start[, end]])`: 返回子串 `sub` 第一次出现的位置, 找不到返回 `-1`。
- `string.index(sub[, start[, end]])`: 类似于 `find()`, 但找不到时抛出异常。
- `string.replace(old, new[, count])`: 替换 `old` 为 `new`, 最多替换 `count` 次。
- `string.count(sub[, start[, end]])`: 统计子串 `sub` 出现的次数。

分割和连接

- `string.split([sep[, maxsplit]])`: 使用 `sep` 分割字符串, 默认按任意空白字符分割。
- `string.rsplit([sep[, maxsplit]])`: 从右开始分割。
- `string.join(iterable)`: 使用字符串作为分隔符, 连接序列中的元素。

检查字符串内容

- `string.startswith(prefix[, start[, end]])`: 检查字符串是否以指定前缀开头。
- `string.endswith(suffix[, start[, end]])`: 检查字符串是否以指定后缀结尾。
- `string.isalpha()`: 是否全是字母。
- `string.isdigit()`: 是否全是数字。
- `string.isalnum()`: 是否全是字母或数字。
- `string.isspace()`: 是否全是空白字符。
- `string.islower()`: 是否全为小写字母。
- `string.isupper()`: 是否全为大写字母。
- `string.istitle()`: 是否为标题格式 (每个单词首字母大写)。

- `len(string)`: 获取字符串长度。
- `string.partition(sep)`: 将字符串分为三部分: `sep` 左边、`sep` 本身、`sep` 右边。
- `string.rpartition(sep)`: 从右边开始查找 `sep` 并分割。

添加和移除元素

- `s.add(element)`: 添加一个元素到集合中。
- `s.update(iterable)`: 添加多个元素到集合中。
- `s.remove(element)`: 移除指定元素, 如果元素不存在则抛出 `KeyError`。
- `s.discard(element)`: 移除指定元素, 如果元素不存在则不报错。
- `s.pop()`: 移除并返回集合中的任意一个元素, 集合为空时抛出 `KeyError`。
- `s.clear()`: 清空集合中的所有元素。

集合

集合操作

- `s.union(t)` 或 `s | t`: 返回两个集合的并集。
- `s.intersection(t)` 或 `s & t`: 返回两个集合的交集。
- `s.difference(t)` 或 `s - t`: 返回在 `s` 中但不在 `t` 中的元素。
- `s.symmetric_difference(t)` 或 `s ^ t`: 返回在 `s` 或 `t` 中, 但不同时在两个集合中的元素。
- `s.copy()`: 返回集合的一个浅拷贝。

检查成员关系

- `element in s`: 检查元素是否存在于集合中。
- `element not in s`: 检查元素是否不存在于集合中。

子集和超集检查

- `s.issubset(t)` 或 `s <= t`: 检查 `s` 是否是 `t` 的子集。
- `s.issuperset(t)` 或 `s >= t`: 检查 `s` 是否是 `t` 的超集。
- `s.isdisjoint(t)`: 检查 `s` 和 `t` 是否没有交集。

获取集合信息

- `len(s)`: 获取集合中元素的数量。
- `min(s)`: 获取集合中的最小元素。
- `max(s)`: 获取集合中的最大元素。
- `sum(s)`: 计算集合中所有元素的总和 (仅适用于数值类型)。


创建列表

- `[]`: 创建一个空列表。
- `[element1, element2, ...]`: 使用方括号创建一个包含指定元素的列表。
- `list(iterable)`: 从可迭代对象 (如字符串、元组等) 创建列表。

添加和移除元素

- `lst.append(element)`: 在列表末尾添加一个元素。
- `lst.extend(iterable)`: 将可迭代对象中的所有元素添加到列表末尾。
- `lst.insert(index, element)`: 在指定位置插入一个元素。
- `lst.remove(element)`: 移除第一个匹配的元素, 如果元素不存在则抛出 `ValueError`。
- `lst.pop([index])`: 移除并返回指定位置的元素, 默认移除最后一个元素。
- `lst.clear()`: 清空列表中的所有元素。

访问和修改元素

- `lst[index]`: 访问或修改指定索引处的元素。
- `lst[start:end:step]`: 切片访问, 获取子列表。
- `lst.index(element[, start[, end]])`: 返回第一个匹配元素的索引, 找不到时抛出 `ValueError`。
- `lst.count(element)`: 统计元素出现的次数。
- `lst.reverse()`: 就地反转列表。
- `lst.sort(key=None, reverse=False)`: 就地对列表进行排序。
- `sorted(lst, key=None, reverse=False)`: 返回排序后  列表, 不改变原列表。

列表

- `lst.copy()` : 返回列表的一个浅拷贝。
- `lst[:]` : 使用切片操作创建列表的浅拷贝。

创建字典

- `{}` : 创建一个空字典。
- `{key1: value1, key2: value2, ...}` : 使用键值对创建字典。
- `dict()` : 创建一个空字典或从可迭代对象创建字典。
- `dict.fromkeys(iterable[, value])` : 使用可迭代对象中的元素作为键, 所有键的值为指定值 (默认为 `None`) 。

添加和修改项

- `d[key] = value` : 添加或更新指定键的值。
- `d.update([other_dict | iterable_of_pairs])` : 更新字典, 可以是另一个字典或键值对的可迭代对象。

访问项

- `d[key]` : 获取指定键对应的值, 如果键不存在则抛出 `KeyError` 。
- `d.get(key[, default])` : 获取指定键对应的值, 如果键不存在则返回默认值 (默认为 `None`) 。
- `d.setdefault(key[, default])` : 如果键存在返回其值; 如果不存在插入键并设置为默认值 (默认为 `None`) , 然后返回该值。

删除项

- `del d[key]` : 删除指定键的项, 如果键不存在则抛出 `KeyError` 。
- `d.pop(key[, default])` : 移除并返回指定键的值, 如果键不存在则返回默认值 (或抛出 `KeyError`) 。
- `d.popitem()` : 移除并返回一个任意的 `(key, value)` 对, 字典为空时抛出 `KeyError` 。
- `d.clear()` : 清空字典中的所有项。



遍历字典

- `for key in d` : 遍历字典的键。
- `for value in d.values()` : 遍历字典的值。
- `for key, value in d.items()` : 遍历字典的键值对。

检查成员关系

- `key in d` : 检查键是否存在于字典中。
- `key not in d` : 检查键是否不存在于字典中。

获取字典信息

- `len(d)` : 获取字典中键值对的数量。
- `list(d.keys())` : 返回包含所有键的列表。
- `list(d.values())` : 返回包含所有值的列表。
- `list(d.items())` : 返回包含所有键值对的列表 (每个键值对是一个元组) 。

复制字典

- `d.copy()` : 返回字典的一个浅拷贝。
- `dict(d)` : 使用构造函数创建字典的浅拷贝。

```
my_list = ['apple', 'banana', 'cherry', 'banana']
indices = [i for i, x in enumerate(my_list) if x == 'banana']
print(indices) # 输出将是 [1, 3], 因为 'banana' 在索引1和3处。
```

math

常量

- `math.pi` : 圆周率 π (约等于 3.141592653589793)。
- `math.e` : 自然对数的底 e (约等于 2.718281828459045)。
- `math.tau` : $\tau = 2\pi$ (约等于 6.283185307179586)。
- `math.inf` : 正无穷大。
- `math.nan` : 非数字 (Not a Number) 。

数值操作

- `math.ceil(x)` : 返回不小于 x 的最小整数。
- `math.floor(x)` : 返回不大于 x 的最大整数。
- `math.trunc(x)` : 截断 x 的小数部分, 返回整数部分。
- `math.fabs(x)` : 返回 x 的绝对值。
- `math.copysign(x, y)` : 返回与 y 同号的 x 。
- `math.gcd(a, b)` : 返回 a 和 b 的最大公约数。
- `math.lcm(a, b)` : 返回 a 和 b 的最小公倍数 (Python 3.9+) 。
- `math.isclose(a, b, *, rel_tol=1e-09, abs_tol=0.0)` : 判断两个浮点数是否接近。

幂和对数

- `math.pow(x, y)` : 返回 x 的 y 次幂。
- `math.sqrt(x)` : 返回 x 的平方根。
- `math.exp(x)` : 返回 e 的 x 次幂。
- `math.log(x[, base])` : 返回 x 的自然对数或以指定 $base$ 为底的对数。
- `math.log2(x)` : 返回 x 的以 2 为底的对数。
- `math.log10(x)` : 返回 x 的以 10 为底的对数。

三角函数

- `math.sin(x)` : 返回 x 弧度的正弦值。
- `math.cos(x)` : 返回 x 弧度的余弦值。
- `math.tan(x)` : 返回 x 弧度的正切值。
- `math.asin(x)` : 返回 x 的反正弦值 (弧度) 。
- `math.acos(x)` : 返回 x 的反余弦值 (弧度) 。
- `math.atan(x)` : 返回 x 的反正切值 (弧度) 。
- `math.atan2(y, x)` : 返回 y/x 的反正切值 (弧度) , 考虑象限。

其他功能

- `math.factorial(x)` : 返回 x 的阶乘。
- `math.isinf(x)` : 如果 x 是正无穷大或负无穷大, 返回 `True`。
- `math.isnan(x)` : 如果 x 是 `NaN`, 返回 `True`。
- `math.modf(x)` : 返回 x 的小数部分和整数部分。
- `math.frexp(x)` : 返回 x 的尾数和指数, $x = m * 2^{**}e$ 。
- `math.ldexp(x, i)` : 返回 $x * 2^{**}i$ 。

```
from functools import lru_cache

@lru_cache(maxsize=128)
```

缓存函数结果

```
def sieve(limit):
    is_prime = [True] * (limit + 1)
    is_prime[0] = is_prime[1] = False
    for i in range(2, int(math.sqrt(limit)) + 1):
        if is_prime[i]:
            for j in range(i * i, limit + 1, i):
                is_prime[j] = False
    #return [i for i in range(limit + 1) if is_prime[i]]
    return is_prime
```

筛法生成素数

```
sys.setrecursionlimit(new_limit)
```

增加递归深度

```
board=[]
board.append( [0 for x in range(m+2)] )
for y in range(n):
    board.append([0] +[int(x) for x in input().split()] + [0])

board.append( [0 for x in range(m+2)] )
```

为矩阵加保护圈

1. 获取日历信息

日期

- `calendar.month(year, month[, w[, l]])` :
 - 返回指定年月的日历字符串表示，其中 `w` 是每行的宽度（默认2），`l` 是每行的高度（默认1）。
- `calendar.calendar(year[, w=2[, l=1[, c=6]])` :
 - 返回指定年的多个月份的日历字符串表示，其中 `w`, `l`, `c` 分别是每周的宽度、每行的高度和每个月之间的列数。

2. 判断闰年

- `calendar.isleap(year)` :
 - 如果给定年份是闰年，则返回 `True`，否则返回 `False`。
- `calendar.leapdays(y1, y2)` :
 - 返回从 `y1` 到 `y2`（不包括 `y2`）之间的闰年天数。

3. 获取星期几

- `calendar.weekday(year, month, day)` :
 - 返回给定日期对应的星期几（0 = Monday, 6 = Sunday）。
- `calendar.firstweekday()` :
 - 返回当前设置的每周第一天，默认是 0（星期一）。可以使用 `calendar.setfirstweekday()` 修改。

1. Counter

collections

- **简介**：用于计数可哈希对象的出现次数。
- **常见方法**：
 - `Counter(iterable)` 或 `Counter(dict)`：创建一个新的 `Counter` 对象。
 - `elements()`：返回一个迭代器，可以遍历所有元素。
 - `most_common([n])`：返回前 `n` 个最常见的元素及其计数。
 - `subtract([iterable-or-mapping])`：从计数中减去元素。

2. defaultdict

- **简介**：类似于常规字典，但可以通过默认工厂函数为不存在的键提供默认值。
- **常见用法**：
 - `defaultdict(default_factory[, ...])`：创建一个新的 `defaultdict` 对象，默认工厂可以是 `list`, `int`, `set` 等。
 - 当访问不存在的键时，会调用默认工厂函数生成默认值。

3. deque

- **简介**：双端队列，支持从两端快速添加或弹出元素。
- **常见方法**：
 - `deque([iterable[, maxlen]])`：创建一个新的 `deque` 对象。
 - `append(x)`：在右边添加一个元素。
 - `appendleft(x)`：在左边添加一个元素。
 - `pop()`：移除并返回最右边的元素。
 - `popleft()`：移除并返回最左边的元素。
 - `extend(iterable)`：在右边扩展多个元素。
 - `extendleft(iterable)`：在左边扩展多个元素。

dfs模板

```
1 #1.dfs
2 import sys
3
4 # input = sys.stdin.read
5 sys.setrecursionlimit(20000)
6
7
8 def dfs(x, y):
9     # 标记当前位置为已访问
10    field[x][y] = '.'
11    # 遍历8个方向
12    for dx, dy in directions:
13        nx, ny = x + dx, y + dy
14        # 检查新位置是否在地图范围内且未被访问
15        if 0 <= nx < n and 0 <= ny < m and field[nx][ny] == 'W':
16            dfs(nx, ny)
17
18
19 # 一次性读取所有输入
20 # data = input().split()
21 # n, m = map(int, data[:2])
22 # field = [list(row) for row in data[2:2+n]]
23 n, m = map(int, input().split())
24 field = [list(input()) for _ in range(n)]
25 # 初始化8个方向
26 directions = [(-1, -1), (-1, 0), (-1, 1), (0, -1), (0, 1), (1, -1), (1, 0), (1, 1)]
27 # 计数器
28 cnt = 0
29
30 # 遍历地图
31 for i in range(n):
32     for j in range(m):
33         if field[i][j] == 'W':
34             dfs(i, j)
35             cnt += 1
36
37 print(cnt)
38
```

```
def binary_search(arr, target):
    left, right = 0, len(arr) - 1

    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid # 返回目标元素的索引
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1

    return -1 # 如果未找到目标元素, 返回 -1
```

```
lo, hi = 0, l+1
ans = -1
while lo < hi:
    mid = (lo + hi) // 2

    if check(mid):
        hi = mid
    else:
        ans = mid
        lo = mid + 1

#print(lo-1)
print(ans)
```

```
def binary_search_solution(func, low, high, tolerance=1e-7):
```

"""

使用二分查找算法求解方程 $\text{func}(x) = 0$ 在 $[\text{low}, \text{high}]$ 区间内的根。

参数:

func : function

需要求解的方程。

low : float

搜索区间的下界。

high : float

搜索区间的上界。

tolerance : float, optional

精度, 默认为 $1e-7$ 。

返回:

float: 方程在给定区间内的根, 精度达到 **tolerance**。

"""

```
if func(low) * func(high) >= 0:
```

```
    print("二分法不能保证收敛: f(low) 和 f(high) 应该有不同的符号")
```

```
    return None
```

```
while (high - low) > tolerance:
```

```
    mid = (low + high) / 2.0
```

```
    if func(mid) == 0:
```

```
        return mid # 找到了精确解
```

```
    elif func(mid) * func(low) < 0:
```

```
        high = mid # 解在左半部分
```

```
    else:
```

```
        low = mid # 解在右半部分
```

```
return (low + high) / 2.0
```



冒泡排序

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        # 标记是否发生了交换
        swapped = False
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                # 交换元素
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True
        # 如果没有发生交换, 说明数组已经排序完成
        if not swapped:
            break
    return arr
```

插入排序

下面是一个改进的插入排序版本:

```
def InsertSort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
    return arr
```

3.5 归并排序(Merge Sort)

归并排序采用分治法, 将待排序数组分成若干个子序列, 分别进行排序, 然后再合并已排序的子序列, 直到整个序列都排好序为止。

代码思路:

1. 将待排序数组分成左右两个子序列, 递归地对左右子序列进行归并排序。
2. 将两个已排序的子序列合并成一个有序序列。

```
def MergeSort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = MergeSort(arr[:mid])
    right = MergeSort(arr[mid:])
    return merge(left, right)

def merge(left, right):
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result

if __name__ == "__main__":
    arr_in = [6, 5, 18, 2, 16, 15, 19, 13, 10, 12, 7, 9, 4, 4, 8, 1, 11, 14, 3, 20, 17, 10]
    print(arr_in)
    arr_out = MergeSort(arr_in)
    print(arr_out)
```



背包问题

因此实际核心代码为

0,1背包

C++ Python

```
1 for i in range(1, n + 1):
2     for l in range(W, w[i] - 1, -1):
3         f[l] = max(f[l], f[l - w[i]] + v[i])
```

完全背包

```
1 for i in range(1, n + 1):
2     for l in range(0, W - w[i] + 1):
3         f[l + w[i]] = max(f[l] + v[i], f[l + w[i]])
```



二进制分组代码

二进制分组简化



C++ Python

```
1 index = 0
2 for i in range(1, m + 1):
3     c = 1
4     p, h, k = map(int, input().split())
5     while k > c:
6         k -= c
7         index += 1
8         list[index].w = c * p
9         list[index].v = c * h
10    c *= 2
11    index += 1
12    list[index].w = p * k
13    list[index].v = h * k
```

混合

```
1 for (循环物品种类) {
2     if (是 0 - 1 背包)
3         套用 0 - 1 背包代码;
4     else if (是完全背包)
5         套用完全背包代码;
6     else if (是多重背包)
7         套用多重背包代码;
8 }
```

「Luogu P1757」通天之分组背包

有 n 件物品和一个大小为 m 的背包，第 i 个物品的价值为 w_i ，体积为 v_i 。同时，每个物品属于一个组，同组内最多只能选择一个物品。求背包能装载物品的最大总价值。

这种题怎么想呢？其实是从「在所有物品中选择一件」变成了「从当前组中选择一件」，于是就对每一组进行一次 0-1 背包就可以了。

再说一说如何进行存储。我们可以将 $t_{k,i}$ 表示第 k 组的第 i 件物品的编号是多少，再用 cnt_k 表示第 k 组物品有多少个。

C++ Python

```
1 for k in range(1, ts + 1): # 循环每一组
2     for i in range(m, -1, -1): # 循环背包容量
3         for j in range(1, cnt[k] + 1): # 循环该组的每一个物品
4             if i >= w[t[k][j]]: # 背包容量充足
5                 dp[i] = max(
6                     dp[i], dp[i - w[t[k][j]]] + c[t[k][j]]
7                 ) # 像0-1背包一样状态转移
```

有依赖的背包

「Luogu P1064」金明的预算方案

金明有 n 元钱，想要买 m 个物品，第 i 件物品的价格为 v_i ，重要度为 p_i 。有些物品是从属于某个主件物品的附件，要买这个物品，必须购买它的主件。

目标是让所有购买的物品的 $v_i \times p_i$ 之和最大。

考虑分类讨论。对于一个主件和它的若干附件，有以下几种可能：只买主件，买主件 + 某些附件。因为这几种可能性只能选一种，所以可以将这看成分组背包。

背包的结果

Python

浅色版本 | 图标

```
# 初始化变量 v 为背包的总容量 V
v = V # 记录当前的存储空间

# 因为最后一件物品存储的是最终状态，所以从最后一件物品进行循环
for i in range(n, 0, -1): # 假设 n 是物品的数量，从最后一个物品开始循环到第一个
    if g[i][v] != g[i-1][v]: # 如果这个条件成立，意味着选择了第 i 项物品
        print(f"选了第 {i} 项物品") # 输出选择了第 i 项物品的信息
        v -= weights[i] # 减去第 i 项物品的重量，更新剩余容量
    else:
        print(f"未选第 {i} 项物品") # 输出未选择第 i 项物品的信息
```

求最优方案总数 ¶

要求最优方案总数，我们要对 0-1 背包里的 dp 数组的定义稍作修改，DP 状态 $f_{i,j}$ 为在只能放前 i 个物品的情况下，容量为 j 的背包「正好装满」所能达到的最大总价值。

这样修改之后，每一种 DP 状态都可以用一个 $g_{i,j}$ 来表示方案数。

$f_{i,j}$ 表示只考虑前 i 个物品时背包体积「正好」是 j 时的最大价值。

$g_{i,j}$ 表示只考虑前 i 个物品时背包体积「正好」是 j 时的方案数。

转移方程：

如果 $f_{i,j} = f_{i-1,j}$ 且 $f_{i,j} \neq f_{i-1,j-v} + w$ 说明我们此时不选择把物品放入背包更优，方案数由 $g_{i-1,j}$ 转移过来，

如果 $f_{i,j} \neq f_{i-1,j}$ 且 $f_{i,j} = f_{i-1,j-v} + w$ 说明我们此时选择把物品放入背包更优，方案数由 $g_{i-1,j-v}$ 转移过来，

如果 $f_{i,j} = f_{i-1,j}$ 且 $f_{i,j} = f_{i-1,j-v} + w$ 说明放入或不放入都能取得最优解，方案数由 $g_{i-1,j}$ 和 $g_{i-1,j-v}$ 转移过来。

背包的第 k 优解

普通的 0-1 背包是要求最优解，在普通的背包 DP 方法上稍作改动，增加一维用于记录当前状态下的前 k 优解，即可得到求 0-1 背包第 k 优解的算法。具体来讲： $dp_{i,j,k}$ 记录了前 i 个物品中，选择的物品总体积为 j 时，能够得到的第 k 大的价值和。这个状态可以理解为将普通 0-1 背包只用记录一个数据的 $dp_{i,j}$ 扩展为记录一个有序的优解序列。转移时，普通背包最优解的求法是

$dp_{i,j} = \max(dp_{i-1,j}, dp_{i-1,j-v_i} + w_i)$ ，现在我们则是要合并 $dp_{i-1,j}$ ， $dp_{i-1,j-v_i} + w_i$ 这两个大小为 k 的递减序列，并保留合并后前 k 大的价值记在 $dp_{i,j}$ 里，这一步利用双指针法，复杂度是 $O(k)$ 的，整体时间复杂度为 $O(nmk)$ 。空间上，此方法与普通背包一样可以压缩掉第一维，复杂度是 $O(mk)$ 的。

- 对于每一个容量 j 和每一个最优解等级 p ，我们检查是否可以选择当前物品 i （即 $j \geq c[i]$ ）。
- 如果可以，我们将 $dp[j - c[i]][p] + w[i]$ 和 $dp[j][p]$ 放入临时列表 a 和 b 中，分别代表选择和不选择当前物品的结果。
- 接着，我们合并这两个列表并去除重复项，同时保持结果的排序顺序。
- 最终，我们将合并后的列表中的第 p 个元素赋值给 $dp[j][p]$ 。

区间dp

「NOI1995」石子合并

题目大意：在一个环上有 n 个数 a_1, a_2, \dots, a_n ，进行 $n-1$ 次合并操作，每次操作将相邻的两堆合并成一堆，能获得新的一堆中的石子数量的和的得分。你需要最大化你的得分。

需要考虑不在环上，而在一条链上的情况。

令 $f(i, j)$ 表示将区间 $[i, j]$ 内的所有石子合并到一起的最大得分。

写出 **状态转移方程**： $f(i, j) = \max\{f(i, k) + f(k+1, j) + \sum_{t=i}^j a_t\} (i \leq k < j)$

令 sum_i 表示 a 数组的前缀和，状态转移方程变形为

$f(i, j) = \max\{f(i, k) + f(k+1, j) + sum_j - sum_{i-1}\}$ 。

方法二：我们将这条链延长两倍，变成 $2 \times n$ 堆，其中第 i 堆与第 $n+i$ 堆相同，用动态规划求解后，取 $f(1, n), f(2, n+1), \dots, f(n, 2n-1)$ 中的最优值，即为最后的答案。时间复杂度 $O(n^3)$ 。

```
1 for len in range(2, n + 1):
2     for i in range(1, 2 * n - len + 1):
3         j = len + i - 1
4         for k in range(i, j):
5             f[i][j] = max(f[i][j], f[i][k] + f[k + 1][j] + sum[j] - sum[i - 1])
```

1. 按位与 (&) :

- 按位与运算符会逐位比较两个操作数，如果相应的两位都为1，则结果为1，否则为0。
- 例如：5 & 3 的二进制表示是 0101 & 0011，结果是 0001，即十进制的1。

2. 按位或 (|) :

- 按位或运算符会逐位比较两个操作数，如果相应的两位中至少有一个为1，则结果为1，否则为0。
- 例如：5 | 3 的二进制表示是 0101 | 0011，结果是 0111，即十进制的7。

3. 按位异或 (^) :

- 按位异或运算符会逐位比较两个操作数，如果相应的两位不同，则结果为1，否则为0。
- 例如：5 ^ 3 的二进制表示是 0101 ^ 0011，结果是 0110，即十进制的6。

4. 按位非 (~) :

- 按位非运算符会对单个操作数的每一位取反，即将1变为0，将0变为1。
- 例如：~5 的二进制表示是 ~0101，结果是 ...11111010（前面省略的是符号位扩展，取决于具体的数据类型）。

5. 左移 (<<) :

- 左移运算符将操作数的所有位向左移动指定的位数，右边用0填充。对于正整数，相当于乘以2的幂。
- 例如：5 << 1 的二进制表示是 0101 << 1，结果是 1010，即十进制的10。

6. 右移 (>>) :

- 右移运算符将操作数的所有位向右移动指定的位数。对于无符号数或正数，左边用0填充；对于负数，左边用1填充（这取决于具体的实现和编程语言）。
- 例如：5 >> 1 的二进制表示是 0101 >> 1，结果是 0010，即十进制的2。



Python

```
from collections import defaultdict
```

```
def dfs(x, num, cur):
    global cnt
    if cur >= n:
        cnt += 1
        sit[cnt] = x
        sta[cnt] = num
        return
    dfs(x, num, cur + 1) # cur位置不放国王
    if cur + 2 <= n: # 确保不会越界
        dfs(x + (1 << cur), num + 1, cur + 2) # cur位置放国王
```

```
def compatible(j, x):
    return not (sit[j] & sit[x]) and \
           not ((sit[j] << 1) & sit[x]) and \
           not (sit[j] & (sit[x] << 1))

n, k = map(int, input().split())
cnt = 0
sit = {}
sta = {}
f = defaultdict(lambda: defaultdict(lambda: defaultdict(int)))
```

```
dfs(0, 0, 0) # 先预处理一行的所有合法状态

# 初始化第一行的状态转移
for j in range(1, cnt + 1):
    f[1][j][sta[j]] = 1

# 动态规划计算所有可能的放置方法
for i in range(2, n + 1):
    for j in range(1, cnt + 1):
        for x in range(1, cnt + 1):
            if not compatible(j, x):
                continue
            for l in range(sta[j], k + 1):
                f[i][j][l] += f[i - 1][x][l - sta[j]]

# 计算并输出答案
ans = sum(f[n][i][k] for i in range(1, cnt + 1))
print(ans)
```

[SCOI2005] 互不侵犯

在 $N \times N$ 的棋盘里面放 K 个国王 ($1 \leq N \leq 9, 1 \leq K \leq N \times N$)，使他们互不攻击，共有多少种摆放方案。国王能攻击到它上下左右，以及左上右下右上左下八个方向上附近的各一个格子，共 8 个格子。

[POI2004] PRZ

有 n 个人需要过桥，第 i 的人的重量为 w_i ，过桥用时为 t_i 。这些人过桥时会分成若干组，只有在某一组的所有人全部过桥后，其余的组才能过桥。桥最大承重为 W ，问这些人全部过桥的最短时间。

$100 \leq W \leq 400, 1 \leq n \leq 16, 1 \leq t_i \leq 50, 10 \leq w_i \leq 100$ 。

```
def solve(n, W, weights, times):
    # 初始化dp数组
    dp = defaultdict(lambda: float('inf'))
    dp[0] = 0

    # 遍历所有可能的状态
    for mask in range(1 << n):
        # 当前状态的总重量
        current_weight = 0
        # 当前状态的最短时间
        current_time = dp[mask]

        # 尝试加入每个人
        for i in range(n):
            if not (mask & (1 << i)):
                # 如果这个人还没有过桥
                if current_weight + weights[i] <= W:
                    # 如果加入这个人后总重量不超过W
                    new_mask = mask | (1 << i)
                    new_time = max(current_time, times[i])
                    dp[new_mask] = min(dp[new_mask], new_time)
                    current_weight += weights[i]
                    current_time = new_time

    # 返回所有人都过桥的最短时间
    return dp[(1 << n) - 1]
```

```
import heapq
```

```
def dijkstra(graph, start):
    # 初始化距离字典，设置起点到自己的距离为0，其余为无穷大
    distances = {vertex: float('infinity') for vertex in graph}
    distances[start] = 0
    # 使用最小堆来存储待处理的节点及其当前已知的最短距离
    priority_queue = [(0, start)]

    while priority_queue:
        current_distance, current_vertex = heapq.heappop(priority_queue)

        # 节点已经被移除的情况
        if current_distance > distances[current_vertex]:
            continue

        for neighbor, weight in graph[current_vertex].items():
            distance = current_distance + weight

            # 只有当找到更短路径时才进行更新
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(priority_queue, (distance, neighbor))

    return distances
```

优先队列

堆

```
1. heappush(heap, item)
```

- 将 `item` 添加到堆 `heap` 中，并保持堆的性质。

```
2. heappop(heap)
```

- 弹出并返回堆中的最小元素（堆顶元素）。如果堆为空，则抛出 `IndexError`。

```
3. heappushpop(heap, item)
```

- 先将 `item` 推入堆中，然后弹出并返回堆中的最小元素。比单独调用 `heappush` 和 `heappop` 更高效。

```
4. heapreplace(heap, item)
```

- 弹出并返回堆中的最小元素，然后将 `item` 推入堆中。与 `heappushpop` 类似，但先弹后推，适用于替换堆顶元素的情况。

```
5. heapify(x)
```

- 将列表 `x` 转换为一个堆，原地进行，时间复杂度 $O(n)$ 。

```
6. nlargest(n, iterable[, key])
```

- 返回可迭代对象 `iterable` 中 n 个最大的元素，可以指定一个 `key` 函数来定制排序逻辑。

```
7. nsmallest(n, iterable[, key])
```

- 返回可迭代对象 `iterable` 中 n 个最小的元素，同样可以指定 `key` 函数。

```
8. merge(*iterables, key=None, reverse=False) (Python 3.5+)
```

- 合并多个已排序的输入流为一个已排序的输出流。`key` 函数用于自定义排序键，`reverse` 参数用于反转排序顺序。



懒删除

需要删除一个元素时先对其打上标记，等到操作到该元素时再将其弹出去（由于一般list中删除的复杂度是线性的，操作时跳过了打了标记的元素；但这种方法在heap中的运用更为典型）。打上标记后元素仍在序列中，但我们将将其**视作已经不存在**进行操作；等到需要真正操作该元素（该元素已到堆顶）时，再做实质上的删除，这样避免了从heap中直接删除元素。

```
#每次要删除x时out[x]+=1
from heapq import heappop, heappush
while ls:
    x = heappop(ls)
    if not out[x]:
        new_min = x
        heappush(ls, x) #不需要弹出的，记得压回去
        break
    out[x]-=1
```


3. combinations(iterable, r)

- **描述:** 返回长度为 `r` 的子序列的所有组合（不重复）。
- **示例:**

Python

浅色版本 ▾ | 

```
print(list(it.combinations('ABCD', 2))) # [('A', 'B'), ('A', 'C'), ('A', 'D'), ('B', 'C'), ('B', 'D'), ('C', 'D')]
```

4. combinations_with_replacement(iterable, r)

- **描述:** 返回长度为 `r` 的子序列的所有组合（允许重复）。
- **示例:**

Python

浅色版本 ▾ | 

```
print(list(it.combinations_with_replacement('AB', 2))) # [('A', 'A'), ('A', 'B'), ('B', 'B')]
```

Python

浅色版本 ▾ | 

```
import itertools

letters = ['A', 'B']
numbers = [1, 2]
symbols = ['!', '@']

combinations = list(itertools.product(letters, numbers, symbols))
print(combinations)
```

输出:

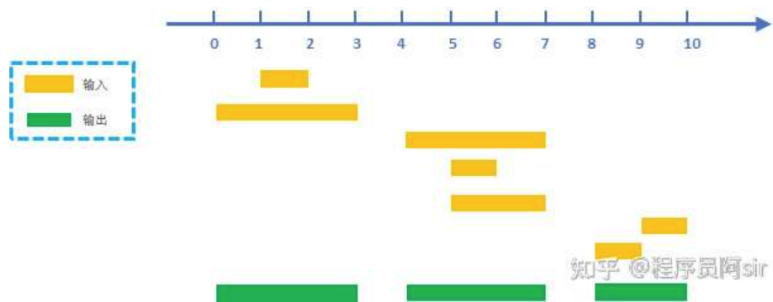
浅色版本 ▾ | 

```
[('A', 1, '!'), ('A', 1, '@'), ('A', 2, '!'), ('A', 2, '@'),
 ('B', 1, '!'), ('B', 1, '@'), ('B', 2, '!'), ('B', 2, '@')]
```

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**。



【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。

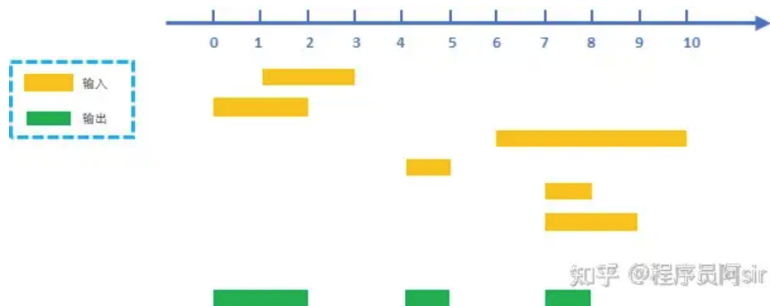
假设当前遍历到的区间为第 i 个区间 $[l_i, r_i]$ ，有以下两种情况：

- $l_i < ed$: 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。



【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。

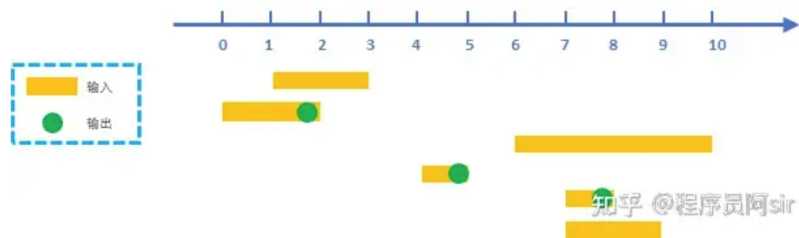
假设当前遍历到的区间为第 i 个区间 $[l_i, r_i]$ ，有以下两种情况：

- $l_i < ed$: 说明当前区间与前面区间有交集。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=[]
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：

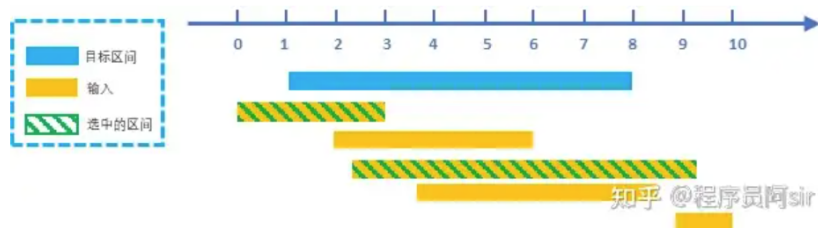
- $l_i < ed$: 说明当前区间与前面区间有交集，前面已经选点了。因此直接跳过。
- $l_i > ed$: 说明当前区间与前面没有交集。因此选中当前区间，并设置 $ed = r_i$ 。

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

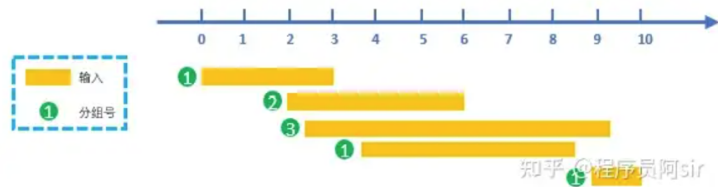
假设右端点最大的区间是第*i*个区间，右端点为 r_i 。

最后将目标区间的start更新成 r_i

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

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)
```

一.函数

输出

`print(f'{ans},{res:.1f})` print是可以带sep和end参数的

字符串

`str.title()`首字母大写（每个单词） `str.lower()/upper()`每个字母小/大写 `str.strip()`去除空格，有`rstrip/lstrip`去掉尾部/头部的空格

`ord()` `chr()` 可以完成字符与ASCII码的转化

`str.find()`查找指定字符，注意如果有的话会返回第一个找到的，如果没有会返回-1

运算

Python的float自然也有精度问题，尽量用int运算，或用//代替/。除法还要注意是否可能出现除以0的情况。舍入时注意round不是严格意义上的四舍五入，遇到恰好.5会向偶数舍入。floor和ceil是安全的。float的等于判断不能用“==”，要用绝对值的差小于某个极小量（或者用math库中的isclose）

math库：最常用的sqrt,对数log(x[,base])、三角sin()、反三角asin()也都有；还有e,pi等常数，inf表示无穷大；返回小于等于x的最大整数floor（），大于等于ceil（），判断两个浮点数是否接近isclose（a，b，*，rel_tol=1e-09,abs_tol=0.0）；一般的取幂pow（x，y），阶乘factorial（x）如果不符合会ValueError,组合数comb（n，k）math.radians()将度数转换为弧度，或者使用math.degrees()将弧度转换为度数。

列表，字典

append以及pop（都是 $O(1)$ 的）。但是注意del，remove，pop(0)，insert，index等都是 $O(n)$ 的！反复remove很有可能导致超时，这里的办法一般是开一个真值表先打标记

切片操作关于所切长度是线性复杂度，反复切片也很可能超时；切片list[k:l]当 $k>=l$ 时不会报错而是返回空列表

in list也是线性复杂度！尽量避免in list的判断，必要时最好用dict或set代替。

list.index()慎用，不仅是线性复杂度，而且在找不到的时候会抛出IndexError

Python特性：允许负数下标，正数越界才会报错。

注意函数有无返回值：list.sort(),list.reverse()都是原地修改而不返回，如要用返回值需用sorted()和reversed()（注意reversed()返回的不是列表而是reversed对象，如需用列表要用list转换类型，但是for循环则不需要转换）

排序list.sort(key=lambda x:x[0],reverse=True) True :3 2 1

list.sort(key=lambda x: (x[0],x[1])) 现根据x[0]排序，再根据x[1]排序

即便如此，你还是不能确定该使用什么样的公式。计算机科学家有时会开玩笑说，那就使用费曼算法（Feynman algorithm）。这个算法是以著名物理学家理查德·费曼命名的，其步骤如下。

- (1) 将问题写下来。
- (2) 好好思考。
- (3) 将答案写下来。