二分查找

```
import bisect
# 找到相同元素的左边的位置/右边的位置
bisect.bisect_left(list,x)
bisect.bisect_right(list,x)
# 插入相同元素的左边/右边
bisect.insort_left(list,x)
bisect.insort_right(list,x)
```

dictionary

```
dict.get("key") # 如果存在，返回值；如果不存在返回None
```

提高递归速度

```
from functools import lru_cache
@lru_cache(maxsize=None) # 但是有可能导致溢出，maxsize可以设置得小一点
```

deque队列

```
import collections
# deque
dq = collections.deque([1, 2, 3])
dq.append(4)
print(dq) # 输出: deque([1, 2, 3, 4])
dq.appendleft(0)
print(dq) # 输出: deque([0, 1, 2, 3, 4])
dq.pop()
print(dq) # 输出: deque([0, 1, 2, 3])
dq.popleft()
print(dq) # 输出: deque([1, 2, 3])
dd = collections.defaultdict(int)
dd['a'] += 1
print(dd) # 输出: defaultdict(<class 'int'>, {'a': 1})
od = collections.OrderedDict()
od['a'] = 1
od['b'] = 2
od['c'] = 3
print(od) # 输出: OrderedDict([('a', 1), ('b', 2), ('c', 3)])
```

优先队列: 即该队列自动保证最小的在第一个（堆顶）

```
import heapq
data = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
heapq.heapify(data)
print(data) # 输出: [0, 1, 2, 3, 9, 5, 4, 6, 8, 7]
heapq.heappush(data, -5) # 最小的元素自动跑到堆顶
print(data) # 输出: [-5, 0, 2, 3, 1, 5, 4, 6, 8, 7, 9]
print(heapq.heappop(data)) # 输出: -5
```

python自带：permutations输出列表元素的全排列

```
from itertools import permutations

def generate_permutations(nums):
    # 使用Python内置的permutations函数生成所有排列
    all_permutations = set(permutations(nums))
    # 将排列转换为列表并排序
    sorted_permutations = sorted(all_permutations)
    return sorted_permutations
```

数字输出格式

```
print(f"{pi:.4f}")  # 输出: 3.1415
print(eval("2+5")) # 输出: 7
```

数据读取和输出

```
import sys
sys.setrecursionlimit(20000) # 防止栈溢出
# 快速读取输入（全部读取）
input = sys.stdin.read
data = input().split()
# 全部粘合为字符串输出
print(" ".join(map(str,results)))
# 没有数据结尾符号的数据
while True:
  try:
    ...
  except EOFError:
    break
```

Dilworth定理（eg.跳高）：

```
# Dilworth定理表明，任何一个有限偏序集的最长反链（无法互相比较大小的一组元素）的长度，
# 等于将该偏序集划分为尽量少的链(即一组具有互相的大小关系的元素)的最小数量。
# 这道题目里，链就是一段段非递减的子序列，反链就是一段递减的子序列
# 最少能用多少个链覆盖list（最少需要几个tester） = 最长的反链的长度（最长的递减子序列）
```

```python
from bisect import bisect_left
def min_testers_needed(scores):
    scores.reverse()  # 反转序列，原来要找的是最长的递减子序列，reverse后=最长的递增的子序列
    # 为什么一定要找递增的子序列呢？因为用bisect_left不会有越界的问题，找出的是有没有比当前的这个score更大的尾巴
    lis = []
    # 用于存储最长递增子序列
    # lis里储存的其实是每一个小递减序列的尾巴
    # 如果出现一个数比原来的某个尾巴小，那么就替换掉比它大一点的那个尾巴，成为新的尾巴（即接到这个尾巴所在的递减序列之后）
    # 如果出现的这个数比原来的所有尾巴都大，那么成为一个新的递减序列的开始
    for score in scores:
        pos = bisect_left(lis, score)
        if pos < len(lis):
            lis[pos] = score
        else:
            lis.append(score)

    return len(lis)
```

Kadane算法："最大子数组问题"（在一个一维数组中找元素和最大的连续子数组）

```python
# 最大子矩阵
'''
为了找到最大的非空子矩阵，可以使用动态规划中的Kadane算法进行扩展来处理二维矩阵。
基本思路是将二维问题转化为一维问题：可以计算出从第i行到第j行的列的累计和，
这样就得到了一个一维数组。然后对这个一维数组应用Kadane算法，找到最大的子数组和。
通过遍历所有可能的行组合，我们可以找到最大的子矩阵。
'''
'''
Kadane算法善于解决"最大子数组问题"：
即在一个一维数组中找到一个**连续子数组**，使得这个子数组的元素之和最大。

如果 current_max + num 大于 num，
这意味着将当前元素 num 添加到现有的子数组（由 current_max 表示）中会得到一个更大的和。
因此，我们选择继续扩展现有的子数组。
如果 current_max + num 小于或等于 num，这表示如果我们将当前元素 num 加入到现有的子数组中，新的和不会比单独的 num 更大。换句话说，现有子数组的和已经变得"负累"，它不会对后续的和产生积极影响。
在这种情况下，更好的选择是从当前元素 num 开始一个新的子数组，因为这样有可能找到一个更大的和。
通过这种方式，Kadane算法确保了在每一步都做出最优的选择：
要么扩展当前的最佳子数组，要么从当前元素重新开始，从而最终能够找到整个数组中的最大子数组和。
'''
def kadane(arr):
    # max_end_here 用于追踪到当前元素为止包含当前元素的最大子数组和
    # max_so_far 用于储存迄今为止遇到的最大子数组和
    max_end_here = max_so_far = arr[0]
    for x in arr[1:]:
        # 对于每个新元素，我们决定是开始一个新的子数组（仅包含当前元素x）
        # 还是将当前元素添加到现有的子数组中
        max_end_here = max(x, max_end_here + x)
        max_so_far = max(max_so_far, max_end_here)
```

```
    return max_so_far

def max_submatrix(matrix):
    rows = len(matrix)
    cols = len(matrix[0])
    max_sum = float('-inf')

    for left in range(cols):
        temp = [0] * rows
        for right in range(left,cols):
            # temp 数组实际上代表了当前选择的列范围内的每一行的总和，因此可以被视为一维数组。
            for row in range(rows):
                temp[row] += matrix[row][right]
                # print(temp)
            max_sum = max(max_sum, kadane(temp))
    return max_sum

n = int(input())
nums = []
matrix = []
while len(nums) < n**2:
    nums.extend(input().split())
    nums = list(map(int, nums))
for i in range(n):
    matrix.append(nums[i*n : (i+1)*n])
# matrix=[list(map(int, nums[i * n:(i+1) * n])) for i in range(n)]
print(max_submatrix(matrix))
```

滑动窗口

```
# 无重复字符的最长子串
class Solution:
    def lengthOfLongestSubstring(self, s: str) -> int:
        # 初始化变量
        start = -1  # 当前无重复子串的起始位置的前一个位置
        max_length = 0  # 最长无重复子串的长度
        char_index = {}  # 字典，记录每个字符最近一次出现的位置

        # 遍历字符串
        for i, char in enumerate(s):
            # 如果字符之前出现过（在char_index里）
            # 且上次出现是在这个子串中（位置大于当前无重复子串的起始位置的前一个位置）
            # 那么当前子串里有重复的字符了
            # 应该从当前这个字符开始寻找新的子串
            if char in char_index and char_index[char] > start:
                # 更新起始位置为该字符上次出现的位置
                start = char_index[char]

            # 更新字典中这个字符最后一次出现的位置
            char_index[char] = i
```

```
            # 计算当前无重复子串的长度，并更新最大长度
            current_length = i - start
            max_length = max(max_length, current_length)

        return max_length
```

约瑟夫问题

```
# 先使用pop从列表中取出，如果不符合要求再append回列表，相当于构成了一个圈
def hot_potato(name_list, num):
    queue = []
    for name in name_list:
        queue.append(name)

    while len(queue) > 1:
        for i in range(num):
            queue.append(queue.pop(0))  # O(N)
        queue.pop(0)                    # O(N)
    return queue.pop(0)                 # O(N)

while True:
    n, m = map(int, input().split())
    if {n,m} == {0}:
        break
    monkey = [i for i in range(1, n+1)]
    print(hot_potato(monkey, m-1))
```

排序（字典序从小到大排列，找下m个）

```
# 递增是小的字典序，递减是大的字典序
# 从右向左查找第一个升序对：
# 从序列的末尾开始向前遍历，找到第一个满足 nums[i] < nums[i + 1] 的索引 i。
# 如果找到了这样的 i，则说明 **从 i+1 到末尾是一个非递增序列** 。
# 从右向左查找第一个大于 nums[i] 的元素：
# 在已经找到的非递增序列中（即从 i+1 到末尾），再次从右向左查找第一个满足 nums[j] > nums[i] 的索引
j。
# 交换 nums[i] 和 nums[j]：将这两个位置的元素交换。
# 反转 i+1 到末尾的元素：由于这部分原本是非递增的，反转后会变成非递减，这是为了确保新的排列是所有可能的比
当前排列大的排列中最小的一个。
# 如果没有找到升序对，说明当前排列已经是最大的了（整个序列是非递增的），那么下一个排列应该是最小的排列，可
以通过直接反转整个序列来得到。
def next_permutation(n,nums):
    # Step 1: Find the first ascending pair from the right
    i = n-2
    while i>= 0 and nums[i]>nums[i+1]:
        i -= 1
    # Step 2: Find the first number larger than nums[i] from the right
    # j一定存在，因为nums[i] < nums[i+1]，j至少可以等于i+1
    if i >= 0:
        j = n-1
```

```
        while nums[j] < nums[i]:
            j -= 1
    # Step 3: Swap nums[i] and nums[j]
        nums[i], nums[j] = nums[j], nums[i]
        # Step 4: Reverse the elements after index i
        nums[i + 1:] = reversed(nums[i + 1:])
    else:
        nums.reverse()

m = int(input())
for _ in range(m):
    n, k = map(int, input().split())
    nums = list(map(int, input().split()))
    for times in range(k):
        next_permutation(n,nums)
    print(*nums)
```

滑雪

```
r, c = map(int, input().split())
node = []
# height of each element，并且需要加一个保护圈（四周有高地）

node.append( [100001 for _ in range(c+2)] )
for _ in range(r):
    node.append([100001] +[int(_) for _ in input().split()] + [100001])

node.append( [100001 for _ in range(c+2)] )
# 四个方位dx，dy
# dp用来记录每个格子开始的最长路径长（一开始都是0）
dp = [[0]*(c+2) for _ in range(r+2)]
dx = [-1, 0, 1, 0]
dy = [ 0, 1, 0,-1]
# 如果算过了ij格子，那么就不用再算一次了
def dfs(i,j):
    if dp[i][j]>0:
        return dp[i][j]
# 如果没有遍历过ij格子，那么如果ij格子四周有比它矮的格子（i+dx, j+dy），就可以往下滑，并且ij格子开始的最长路径是1+（i+dx, j+dy）格子出发的最长路径
# 递归调用dfs函数
    for k in range(4):
        if node[i+dx[k]][j+dy[k]] < node[i][j]:
            dp[i][j] = max( dp[i][j], dfs(i+dx[k], j+dy[k])+1 )

    return dp[i][j]
# 所有格子都经历一遍，ans是其中最大的一个路径
ans = 0
for i in range(1, r+1):
    for j in range(1, c+1):
        ans = max( ans, dfs(i,j) )
# 加一是因为，最长的坡道最后还要滑下去一格（就是算数列的长度不是算间隔的多少）
```

```
print(ans+1)
```

水淹七军

```
# 广度优先搜索bfs
from collections import deque

# 判断坐标是否有效
def is_valid(x, y, m, n):
    return 0 <= x < m and 0 <= y < n

# 广度优先搜索模拟水流
def bfs(start_x, start_y, start_height, m, n, h, water_height):
    dx = [-1, 1, 0, 0]
    dy = [0, 0, -1, 1]
    q = deque([(start_x, start_y, start_height)])
    water_height[start_x][start_y] = start_height

    while q:
        x, y, height = q.popleft()
        for i in range(4):
            nx, ny = x + dx[i], y + dy[i]
            if is_valid(nx, ny, m, n) and h[nx][ny] < height:
                if water_height[nx][ny] < height:
                    water_height[nx][ny] = height
                    q.append((nx, ny, height))
# i,j 是司令部的坐标，p个防水点
for _ in range(p):
     if h[x][y] <= h[i][j]:
         continue
     bfs(x, y, h[x][y], m, n, h, water_height)
```

```
# 深度优先搜索dfs
# 深度优先搜索模拟水流
def dfs(x, y, water_height_value, m, n, h, water_height):
    dx = [-1, 1, 0, 0]
    dy = [0, 0, -1, 1]

    for i in range(4):
        nx, ny = x + dx[i], y + dy[i]
        if is_valid(nx, ny, m, n) and h[nx][ny] < water_height_value:
            if water_height[nx][ny] < water_height_value:
                water_height[x][y] = water_height_value
                dfs(nx, ny, water_height_value, m, n, h, water_height)
# i,j 是司令部的坐标，有p个放水点
for _ in range(p):
      if h[x][y] <= h[i][j]:
         continue
      dfs(x, y, h[x][y], m, n, h, water_height)
```

lake counting

```
import sys
sys.setrecursionlimit(20000)
n,m = map(int,input().split())
matrix = [list(input()) for _ in range(n)]
# 八个方向（adjacent）
directions = ((-1,-1),(-1,0),(-1,1),(0,-1),(0,1),(1,-1),(1,0),(1,1))
# 初始化count
count = 0
# 这个dfs函数的逻辑是，只要找到一个W，就把它附近的W全部设置为.
# 因为这样可以标记哪些W已经被计算进同一个湖里了，可以提高效率
def dfs(x,y):
    matrix[x][y] = '.'
    for dx, dy in directions:
        nx, ny = x + dx, y + dy
        if 0 <= nx <n and 0 <= ny <m and matrix[nx][ny] != ".":
            dfs(nx,ny)
# 遍历地图
for i in range(n):
    for j in range(m):
        if matrix[i][j] == 'W':
            dfs(i,j)
            count += 1
print(count)
```