一、图

1.最短路径算法

(1)Dijkstra

```
def dijkstra_two_nodes(graph, start, end):#graph: 邻接表。需要求出路径的。
    #如果存在路径，返回(最短距离，路径列表)，如果不存在路径，返回(float('inf'), [])
    n = len(graph)
    dist = [float('inf')] * n  # 初始化所有节点距离为无穷大
    dist[start] = 0            # 起点到自身的距离为0
    visited = [False] * n      # 记录节点是否已被访问
    prev = [-1] * n            # 记录前驱节点，用于重建路径
    heap = [(0, start)]        # 优先队列，存储(距离，节点)
    while heap:
        d, u = heapq.heappop(heap)  # 取出当前距离最小的节点
        if u == end: # 如果找到目标节点，提前终止并返回结果
            path = [] # 重建路径
            current = end
            while current != -1:
                path.append(current)
                current = prev[current]
            path.reverse()
            return d, path
        if visited[u]:
            continue
        visited[u] = True
        for v, weight in graph[u]:# 遍历当前节点的所有邻居
            if not visited[v]:  # 只考虑未访问的节点
                new_dist = dist[u] + weight
                if new_dist < dist[v]:  # 发现更短路径
                    dist[v] = new_dist
                    prev[v] = u  # 记录前驱节点
                    heapq.heappush(heap, (new_dist, v))  # 将新距离加入堆
    return float('inf'), []
```

```
def dijkstra(graph, start, end): #using distance
    n = len(graph)
    dist = [float('inf')] * n
    dist[start] = 0
    heap = [(0, start)]
    while heap:
        d, u = heapq.heappop(heap)
        if d != dist[u]:
            continue
        if u == end:
            return d
        for v, weight in graph[u]:
            new_dist = d + weight
            if new_dist < dist[v]:
```

```
                dist[v] = new_dist
                heapq.heappush(heap, (new_dist, v))
    return -1

def dijkstra(graph, start, end): # using visited
    n = len(graph)
    visited = [False] * n
    heap = [(0, start)]
    while heap:
        d, u = heapq.heappop(heap)
        if visited[u]:
            continue
        visited[u] = True # 必须在这里设置。
        if u == end:
            return d
        for v, weight in graph[u]:
            if not visited[v]:
                heapq.heappush(heap, (d + weight, v))
    return -1
```

(2)Floyd-Warshall

逐步考虑每个顶点作为中间节点,检查通过该中间节点是否能缩短任意两个顶点之间的路径距离,通过不断更新距离矩阵来找到所有顶点对之间的最短路径。
算法特点：多源最短路径：计算图中所有顶点对之间的最短路径；支持负权边：可以处理图中包含负权边的情况；检测负权环：能够识别图中是否存在负权环。(如果存在任意 i 使得 dist[i][i] < 0，则图中存在负权环。)

```
def floyd_warshall(dist):# dist:邻接矩阵。dist[i][i]=0
    n = len(dist)
    for k in range(n):          # 中间节点
        for i in range(n):      # 起始节点
            for j in range(n):  # 终止节点
                # 如果通过k节点中转的路径更短
                if dist[i][k] + dist[k][j] < dist[i][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]
    return dist
```

2.最小生成树

(1)Prim
1.初始化：选择一个起始顶点，加入生成树。
2.重复直到所有顶点加入：找到连接生成树内顶点和生成树外顶点的最小权值边，将该边及其连接的生成树外顶点加入生成树。O(ElogV)

```
def prim(graph): #graph:邻接表
    visited = [False for _ in range(len(graph))]
    heap = []   # 小顶堆：(权重，当前节点，父节点)
    mst = []    # 最小生成树的边
    start = 0
```

```
    visited[start] = True
    for neighbor, weight in graph[start]:
        heapq.heappush(heap, (weight, start, neighbor))
    while heap:
        weight, u, v = heapq.heappop(heap)
        if visited[v]:
            continue
        visited[v] = True
        mst.append((u, v, weight))
        for neighbor, w in graph[v]:
            if not visited[neighbor]:
                heapq.heappush(heap, (w, v, neighbor))
    return mst
```

(2)Kruskal

将所有边按权值从小到大排序。初始化并查集（用于检测环）。遍历每条边：如果边的两个顶点不在同一集合（不形成环），则加入生成树并合并集合。O(ElogE),并查集

```
def kruskal(graph, n):
    parent = list(range(n))
    rank = [0] * n
    def find(x):
        if parent[x] != x:
            parent[x] = find(parent[x])
        return parent[x]
    def union(x, y):
        rx, ry = find(x), find(y)
        if rx == ry:
            return False
        if rank[rx] < rank[ry]:
            parent[rx] = ry
        elif rank[rx] > rank[ry]:
            parent[ry] = rx
        else:
            parent[ry] = rx
            rank[rx] += 1
        return True
    edges = []# 获取所有边并排序
    for u in range(n):
        for v, w in graph[u]:
            edges.append((w, u, v))
    edges.sort()
    mst = []
    for w, u, v in edges:
        if union(u, v):
            mst.append((u, v, w))
            if len(mst) == n - 1:
                break
    return mst
```

3.强连通分量，Kosaraju(2DFS)

Kosaraju算法是一种用于在有向图中寻找强连通分量（Strongly Connected Components，SCC）的算法。它基于深度优先搜索（DFS）和图的转置操作。核心思想就是两次深度优先搜索（DFS）。

在第一次DFS中，我们对图进行标准的深度优先搜索，但是在此过程中，我们记录下顶点完成搜索的顺序。这一步的目的是为了找出每个顶点的完成时间（即结束时间）。

接下来，我们对原图取反，即将所有的边方向反转，得到反向图。

在第二次DFS中，我们按照第一步中记录的顶点完成时间的逆序，对反向图进行DFS。这样，我们将找出反向图中的强连通分量。

Kosaraju算法的关键在于第二次DFS的顺序，它保证了在DFS的过程中，我们能够优先访问到整个图中的强连通分量。因此，Kosaraju算法的时间复杂度为O(V + E)，其中V是顶点数，E是边数。使用stack模拟按照结束时间的递减顺序访问顶点

```
def dfs1(graph, node, visited, stack):
    visited[node] = True
    for neighbor in graph[node]:
        if not visited[neighbor]:
            dfs1(graph, neighbor, visited, stack)
    stack.append(node)
def dfs2(graph, node, visited, component):
    visited[node] = True
    component.append(node)
    for neighbor in graph[node]:
        if not visited[neighbor]:
            dfs2(graph, neighbor, visited, component)
def kosaraju(graph): # 邻接表
    # Step 1: Perform first DFS to get finishing times
    stack = []
    visited = [False] * len(graph)
    for node in range(len(graph)):
        if not visited[node]:
            dfs1(graph, node, visited, stack)
    # Step 2: Transpose the graph
    transposed_graph = [[] for _ in range(len(graph))]
    for node in range(len(graph)):
        for neighbor in graph[node]:
            transposed_graph[neighbor].append(node)
    # Step 3: Perform second DFS on the transposed graph to find SCCs
    visited = [False] * len(graph)
    sccs = []
    while stack:
        node = stack.pop()
        if not visited[node]:
            scc = []
            dfs2(transposed_graph, node, visited, scc)
            sccs.append(scc)
    return sccs
```

4.拓扑排序(Kahn)

```
def topological_kahn(graph):#邻接表
    in_degree = {node: 0 for node in graph}# 计算入度
    for node in graph:
        for neighbor in graph[node]:
            in_degree[neighbor] = in_degree.get(neighbor, 0) + 1
    queue = deque([node for node in graph if in_degree[node] == 0])# 初始化队列（入
度为0）
    topo_order = []
    while queue:
        node = queue.popleft()
        topo_order.append(node)
        for neighbor in graph.get(node, []):# 更新相邻节点的入度
            in_degree[neighbor] -= 1
            if in_degree[neighbor] == 0:
                queue.append(neighbor)
    if len(topo_order) == len(graph):# 检查是否所有节点都被访问
        return topo_order
    return "图有环，无法拓扑排序"
```

5.求无向图的直径
(1)Floyd-Warshall算法，找出距离矩阵中的最大值即为直径
(2)两次BFS/DFS（适用于无环图/树）从任意节点A出发进行BFS/DFS，找到距离A最远的节点B，从节点B出发进行BFS/DFS，找到距离B最远的节点C，B到C的距离即为图的直径

6.判环
(1)无向图
i.DFS，每次DFS时，记录当前节点的parent，若访问到了已经访问过的节点，且不是parent，说明存在环。

```
def has_cycle_undirected(graph):
    visited = set()
    def dfs(node, parent):
        visited.add(node)
        for neighbor in graph[node]:
            if neighbor not in visited:
                if dfs(neighbor, node):
                    return True
            elif neighbor != parent:
                return True
        return False
    for node in graph:
        if node not in visited:
            if dfs(node, -1):
                return True
    return False
```

ii.并查集。初始每个点属于不同的集合，每条边连接两个点，如果两个点已经在一个集合中，说明成环。适合稠密图，边比较多时效率较高。
(2)有向图:拓扑排序，成功排序则无环。或者：

```
def has_cycle_directed(graph):
    visited = set()
    rec_stack = set()
    def dfs(node):
        visited.add(node)
        rec_stack.add(node)
        for neighbor in graph[node]:
            if neighbor not in visited:
                if dfs(neighbor):
                    return True
            elif neighbor in rec_stack:
                return True
        rec_stack.remove(node)
        return False
    for node in graph:
        if node not in visited:
            if dfs(node):
                return True
    return False
```

7.Warnsdorff算法是一种用于解决骑士巡游问题的启发式算法，贪心策略：在每一步选择具有最少可行出口的下一步位置。

二、树

1.并查集，见Kruskal

2.树转换为二叉树（左孩子-右兄弟表示法，左指针指向第一个孩子，右指针指向下一个兄弟）
转换步骤：将树的根节点作为二叉树的根节点。对于每个节点：将其第一个子节点作为二叉树的左孩子，将其下一个兄弟节点作为二叉树的右孩子。
逆转换：二叉树的根节点作为树的根节点。对于每个节点：其左孩子成为树的第一个子节点，其右孩子成为树的兄弟节点。

3.Huffman编码树。
例题：构造一个具有n个外部节点的扩充二叉树，每个外部节点Ki有一个Wi对应，作为该外部节点的权。使得这个扩充二叉树的叶节点带权外部路径长度总和最小：Min(W1 * L1 + W2 * L2 + W3 * L3 + ... + Wn * Ln)，Wi:每个节点的权值，Li:根节点到第i个外部叶子节点的距离。编程计算最小外部路径长度总和。
输入：第一行输入一个整数n，外部节点的个数。第二行输入n个整数，代表各个外部节点的权值。
输出：输出最小外部路径长度总和。2<=n<=100
样例输入
4
1 1 3 5
样例输出
17

```
import heapq
n = int(input())
weights = list(map(int, input().split()))
heapq.heapify(weights)
```

```
total = 0
while len(weights) > 1:
    a = heapq.heappop(weights)
    b = heapq.heappop(weights)
    s = a + b
    total += s
    heapq.heappush(weights, s)
print(total)
```

4.前序中序转后序

```
def post_order(pre_str, in_str):
    if not pre_str:
        return ""
    root = pre_str[0]
    idx = in_str.find(root)
    left_in = in_str[:idx]
    right_in = in_str[idx+1:]
    left_pre = pre_str[1:1+len(left_in)]
    right_pre = pre_str[1+len(left_in):]
    left_post = post_order(left_pre, left_in)
    right_post = post_order(right_pre, right_in)
    return left_post + right_post + root
```

三、二分搜索

```
left = 1  # 最小值
right = max_possible_value  # 最大值
while left <= right:
    mid = left + (right - left) // 2
    if feasible(mid):  # 检查mid是否可行
        left = mid + 1  # 尝试更大的值
    else:
        right = mid - 1  # 尝试更小的值
    return right  # 最终right是可行的最大值
```

四、排序，冒泡和归并是稳定的。

冒泡排序

```
for i in range(n):
    ok=True
    for j in range(0,n-i-1):
        if arr[j]>arr[j+1]:
            arr[j],arr[j+1]=arr[j+1],arr[j]
            ok=False
```

```
        if ok:
            break
```

快速随机排序

```
def quicksort(arr, left, right):
    if left < right:
        mid = partition(arr, left, right)
        quicksort(arr, left, mid - 1)
        quicksort(arr, mid + 1, right)
def partition(arr, left, right):
    i = left
    j = right - 1
    pivot = arr[right]
    while i <= j:
        while i <= right and arr[i] < pivot:
            i += 1
        while j >= left and arr[j] >= pivot:
            j -= 1
        if i < j:
            arr[i], arr[j] = arr[j], arr[i]
    if arr[i] > pivot:
        arr[i], arr[right] = arr[right], arr[i]
    return i
```

分治排序(归并排序)，用于计算逆序数。

```
def mergeSort(arr):
    if len(arr) <= 1:
        return arr, 0
    mid = len(arr) // 2
    L = arr[:mid]
    R = arr[mid:]
    # 递归排序并计算左右子数组的逆序数
    L, inv_left = mergeSort(L)
    R, inv_right = mergeSort(R)
    # 初始化总逆序数（左右子数组的逆序数之和）
    total_inversions = inv_left + inv_right
    # 合并过程
    i = j = k = 0
    while i < len(L) and j < len(R):
        if L[i] <= R[j]:
            arr[k] = L[i]
            i += 1
        else:
            arr[k] = R[j]
            j += 1
            # 关键：当从右子数组取元素时，左子数组剩余元素都与该元素构成逆序对
            total_inversions += len(L) - i
```

```
            k += 1
        # 处理剩余元素
        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1
        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1
        return arr, total_inversions
```

五、其他

1.Kadane's卡登算法(最大子数组)对于每个元素，只有两种选择：重新开始：以当前元素作为新子数组的起点（和 = num）；延续子数组：将当前元素加入前一个子数组（和 = max_current + num）。选择更优的方案更新max_current。
若是二维的，可以：固定矩阵的上下边界（即选取某些连续的行），将这些行之间同一列的元素相加，形成一个一维数组。对这个一维数组使用Kadane算法，得到当前上下边界内的最大子矩阵和。遍历所有可能的上下边界，记录最大值。

```
def max_subarray_sum(arr):
    if not arr:
        return 0
    max_current=max_global=arr[0]
    for num in arr[1:]:
        max_current =max(num,max_current+num)
        if max_current>max_global:
            max_global= max_current
    return max_global
```

2.调度场算法Shunting Yard Algorithm，中缀表达式转换为后缀表达式（逆波兰表达式）。
调度场算法使用两个数据结构和一个操作符优先级表来处理表达式转换：输出队列：存储最终的后缀表达式，操作符栈：临时存储操作符和括号，优先级表：确定操作符的处理顺序。
(1)初始化，创建空输出队列和空操作符栈。
(2)从左到右扫描输入表达式
遇到操作数：直接加入输出队列
遇到左括号：压入操作符栈
遇到操作符：
当栈顶操作符优先级不低于当前操作符时，弹出栈顶操作符加入输出队列
将当前操作符压入栈
遇到右括号：
弹出操作符栈的元素加入输出队列，直到遇到左括号；
弹出左括号但不加入输出队列
(3)处理剩余操作符，表达式扫描完成后，将栈中剩余操作符依次弹出加入输出队列

```
```

```python
def shunting_yard(expression):#中缀表达式字符串，例如 "3 + 4 * 2 / (1 - 5)^2"
    #返回:后缀表达式列表，例如 ['3','4','2','*','1','5','-','2','^','/','+']
    # 操作符优先级字典,操作符结合性字典
    precedence = { '^': 4,'*': 3,'/': 3,'+': 2,'-': 2}
    associativity = {'^': 'right','*': 'left','/': 'left','+': 'left','-': 'left'}
    output_queue = [] # 输出队列（存储最终的后缀表达式）
    operator_stack = [] # 操作符栈（临时存储操作符和括号）
    tokens = [] # 分词函数：将表达式字符串拆分为tokens
    current_token = ''
    for char in expression:
        if char == ' ': # 遇到空格，结束当前token
            if current_token:
                tokens.append(current_token)
                current_token = ''
        elif char in '()^*/+-': # 遇到操作符或括号
            if current_token:
                tokens.append(current_token)
                current_token = ''
            tokens.append(char)
        else: # 数字或变量名的一部分
            current_token += char
    # 处理最后一个token
    if current_token:
        tokens.append(current_token)
    # 处理每个token
    for token in tokens:
        # 1. 如果是数字或变量，直接加入输出队列
        if token.isalnum() or token.replace('.', '', 1).isdigit():
            output_queue.append(token)
        elif token in precedence: # 2. 如果是操作符
            # 处理栈顶优先级更高或相等的操作符
            while operator_stack and operator_stack[-1] != '(':
                top_op = operator_stack[-1]
                # 比较优先级
                higher_precedence = precedence[top_op] > precedence[token]
                same_precedence = precedence[top_op] == precedence[token]
                left_assoc = associativity[token] == 'left'
                if higher_precedence or (same_precedence and left_assoc):
                    output_queue.append(operator_stack.pop())
                else:
                    break
            operator_stack.append(token) # 当前操作符入栈
        elif token == '(': # 3. 如果是左括号，入栈
            operator_stack.append(token)
        elif token == ')': # 4. 如果是右括号
            # 弹出操作符直到遇到左括号
            while operator_stack and operator_stack[-1] != '(':
                output_queue.append(operator_stack.pop())
            # 如果没有找到左括号，说明括号不匹配
            if not operator_stack or operator_stack[-1] != '(':
                raise ValueError("Mismatched parentheses")
            operator_stack.pop() # 弹出左括号
    while operator_stack: # 处理栈中剩余的操作符
```

```
            op = operator_stack.pop()
            if op == '(':
                raise ValueError("Mismatched parentheses")
            output_queue.append(op)
        return output_queue
```

3.KMP字符串匹配算法，在文本串中匹配模式串
KMP算法在匹配之前，会对模式串 P 进行一个预处理，生成一个部分匹配表，通常称为next数组。
next[j] 的含义： 对于模式串 P 的子串 P[0..j]（即从开头到位置 j 的子串，闭区间），其最长的、相等的、真前缀和真后缀的长度。
真前缀： 不包含最后一个字符的所有前缀（如 "ABC" 的真前缀有 "", "A", "AB"）。
真后缀： 不包含第一个字符的所有后缀（如 "ABC" 的真后缀有 "", "C", "BC"）。
最长相等真前缀和真后缀： 找一个最长的字符串，它既是 P[0..j] 的真前缀，又是 P[0..j] 的真后缀。

算法步骤： 初始化： 文本串指针 i = 0，模式串指针 j = 0。
循环比较： 当 i < len(S) 时：
如果 S[i] == P[j]： 匹配成功！i++, j++。
如果 j == len(P)：找到一个完整匹配！记录位置 i - j。然后利用 next 数组移动模式串：j = next[j-1]（或类似操作，具体实现可能略有不同，目的是跳过已匹配前缀）。
如果 S[i] != P[j]： 发生不匹配！
如果 j > 0： 根据 next 数组移动模式串：j = next[j-1]。文本串指针 i 不动！ 这一步是关键，它利用了 next 数组的信息，跳过了无效匹配位置。
如果 j == 0： 模式串第一个字符就不匹配，只能将文本串指针右移：i++。
结束： 当 i 遍历完文本串，算法结束。

```
def kmp(s1,s2):
    n,m=len(s1),len(s2)
    x,y=0,0
    nt=nextarray(s2,m)
    while x<n and y<m:
        if s1[x]==s2[y]:
            x+=1
            y+=1
        elif y==0:
            x+=1
        else:
            y=nt[y]
    return x-y if y==m else -1
def nextarray(s,m):
    if m==1:
        return [-1]
    nt=[0]*m
    nt[0],nt[1]=-1,0
    i,cn=2,0
    while i<m:
        if s[i-1]==s[cn]:
            cn+=1
            nt[i]=cn
            i+=1
```

```
        elif cn>0:
            cn=nt[cn]
        else:
            nt[i]=0
            i+=1
    return nt
```

4.Manacher算法，求最长回文子串

1.预处理，在字符串中插入特殊分隔符（如#），将任意字符串统一转为奇数长度。例如："aba" → "#a#b#a#"
2.维护一个回文半径数组P，P[i]表示以i为中心的最长回文半径（包含中心）。 3.动态维护最右回文边界，引入两个关键变量：
C：当前已知回文串的中心位置，R：以 C 为中心的回文串的右边界索引（R = C + P[C]）
在遍历过程中动态更新 C 和 R，利用对称性快速计算 P[i]。
遍历 T 的每个位置 i：
若 i <= R：利用对称性，设 i_mirror = 2*C - i（i关于C的对称点），则：P[i] = min(R - i, P[i_mirror])
若 i > R：初始化 P[i] = 0
中心扩展：以 i 为中心向两侧扩展，更新 P[i]
更新 C 和 R：若 i + P[i] > R，则令 C = i，R = i + P[i]
4.结果：在 P 中找到最大值 max_radius 及其中心 center_index

```
def manacher(s):
    ss= '#' + '#'.join(s) + '#'
    n=len(ss)
    p=[0]*n
    ans=0
    c,r=0,0
    for i in range(n):
        length=min(p[2*c-i],r-i) if r>i else 1
        while i+length<n and i-length>=0 and ss[i + length]==ss[i - length]:
            length+=1
        if i+length>r:
            r=i+length
            c=i
        ans=max(ans,length)
        p[i]=length
    return ans-1
```

5.括号嵌套树(stack)根据A(B(E),C(F,G),D(H(I)))的格式建树
不正常做法：

```
class Node:
    def __init__(self, val: str, *children):
        self.val = val
        self.children = [i if isinstance(i,Node) else i() for i in children]
        # 传进来的子树，如果是Node类型的，则还有子子树；如果是function类型的，则没有子子
树了，就是一个节点，需要call一下让它变成节点
for i in 'ABCDEFGHIJKLMNOPQRSTUVWXYZ':
```

```
    exec(f'{i}=lambda *args: Node("{i}",*args)')
root = eval(input()) # 6
if callable(root):# 如果只有一个
    root = root()
```

正常做法：

```
s=input()
where=0
def f(i):
    global where,s
    st=[]
    while i<len(s) and s[i]!=')':
        if s[i]==',':
            i+=1
        elif 'A'<=s[i]<='Z':
            st.append(treenode(s[i]))
            i+=1
        elif s[i]=='(':
            st[-1].children=f(i+1)
            i=where+1
    where=i
    return st
root=f(0)[0]
```

6.双指针
给定一组数S（可能包含相同的数），从中选取两个数，使两数之和离目标数T最近。
输入包含两行。第一行为目标数T。第二行为S中的N个数字，每个数之间以空格隔开。2<=N<=100000
输出离T最近的两数之和。如果存在多个解，则输出数值较小的那个。
样例输入
14
3 7 8 3 9
样例输出
15

```
t=int(input())
nums=sorted(map(int,input().split()))
n=len(nums)
l,r=0,n-1
best=sum(nums)
while l<r:
    s=nums[l]+nums[r]
    if abs(s-t)<abs(best-t) or ( abs(s-t)==abs(best-t) and s<t ):
        best = s
    if s==t:
        break
    elif s<t:
        l+=1
```

```
    else:
        r-=1
print(best)
```

7.单调栈，例题：柱状图中最大的矩形

```
def largestRectangleArea(heights):
    heights.append(0) # 添加哨兵（高度0）确保所有柱子都能被弹出
    stack = [-1]  # 存储索引的栈，初始放入哨兵-1
    max_area = 0
    for i in range(len(heights)):
        # 当前柱子高度小于栈顶柱子高度时
        while stack and heights[i] < heights[stack[-1]]:
            # 弹出栈顶柱子作为矩形高度
            h = heights[stack.pop()]
            # 计算宽度：右边界i - 左边界(新栈顶) - 1
            w = i - stack[-1] - 1
            max_area = max(max_area, h * w)
        # 当前柱子入栈
        stack.append(i)
    return max_area
```

关闭VSCode标红：
1.打开 VSCode 设置：
快捷键 Ctrl + ，（Windows/Linux）或 Cmd + ，（Mac）。
或通过菜单：File > Preferences > Settings。
2.搜索并禁用 Linter：
搜索 Python Linting 或 python.linting.enabled。
取消勾选 **Python > Linting: Enabled**。

常用函数：

```
import heapq
from collections import deque,defaultdict,Counter
import sys # sys.setrecursionlimit(10000)
#读一行sys.stdin.readline() 全读sys.stdin.read()
from functools import lru_cache #@lru_cache(maxsize=None)
import bisect
bisect.bisect_left(a,x,lo=0,hi=len(a))#返回第一个大于等于x的元素位置
bisect.bisect_right(a,x,lo=0,hi=len(a))#返回第一个大于x的元素位置,别名bisect.bisect
bisect.bisect_right(a,6)#返回在a列表中若要插入6的index（有重复数字会插在右边）
bisect.insort(a,6)#返回插入6后的列表a,也有insort_left,insort_right
```

itertools

函数	描述	示例

函数	描述	示例
`product(*iterables, repeat=1)`	笛卡尔积	`product('AB', [1,2]) → (A,1), (A,2), (B,1), (B,2)`
`permutations(iterable, r=None)`	顺序敏感排列	`permutations('AB', 2) → (A,B), (B,A)`
`combinations(iterable, r)`	顺序不敏感组合	`combinations('ABC', 2) → (A,B), (A,C), (B,C)`
`combinations_with_replacement(iterable, r)`	元素可重复组合	`combinations_with_replacement('AB', 2) → (A,A), (A,B), (B,B)`

一般每秒能执行约 10^8 次运算（Python 可能要除以 10），可以据此估计能通过的时间复杂度，如下表所示。

数据范围	允许的时间复杂度	适用算法举例
$n \leq 10$	$O(n!)$ 或 $O(C^n)$	回溯、暴力搜索
$n \leq 20$	$O(2^n)$	状态压缩 DP
$n \leq 40$	$O(2^{n/2})$	折半枚举
$n \leq 10^2$	$O(n^3)$	三重循环的 DP、Floyd
$n \leq 10^3$	$O(n^2)$	二重循环的 DP、背包
$n \leq 10^5$	$O(n \log n)$	各类常用算法
$n \leq 10^6$	$O(n)$	线性 DP、滑动窗口
$n \leq 10^9$	$O(\sqrt{n})$	判断质数
$n \leq 10^{18}$	$O(\log n)$ 或 $O(1)$	二分、快速幂、数学公式