$Sort$

```
list.sort(key=None, reverse=False)
```

key是一个比较函数，默认对对象中的每个元素进行字典序比较，可以替换为自定义函数。

$reverse$是排序规则，$reverse = True$降序，$reverse = False$升序（默认）

```
#比如说对一个元素是list的list进行排序：

data = [[3, 4], [1, 2], [3, 1], [2, 3]]
data.sort(key=lambda x: (x[0], -x[1]))

#表示用来比较的元素是先第一个元素，后第二个元素的相反数，也就是第一个升序，第二个降序
```

当比较方法十分复杂的时候我们可以在$sort$外部定义一个比较函数

```
def custom_sort_key(item):
    unit_weights = {'K': 1, 'M': 2, 'B': 3}
    number_part, unit_part = item[:-1], item[-1]
    unit_weight = unit_weights.get(unit_part, 0)
    return (unit_weight, float(number_part))

data = ["1.23B", "5.67K", "10.5M", "3.21B"]
data.sort(key=custom_sort_key)
```

$Kadane$算法 — 查找最大子数组

$Kadane\ Pro$

```
# OJ-20744:土豪购物
def kadane(list1):
    localmax=globalmax=list1[0]
    for i in list1[1:]:
        localmax=max(localmax+i,i)
        globalmax=max(globalmax,localmax)
    return globalmax

def tuhaokadane(list1):
    ans=kadane(list1)
    leftmax=[]
    rightmax=[]

    sum1=0
    for i in list1:
        sum1=max(sum1+i,i)
        leftmax.append(sum1)

    sum1=0
    for i in list1[::-1]:
        sum1=max(sum1+i,i)
        rightmax.append(sum1)

    rightmax.reverse()
    for i in range(1,len(list1)-1):
        ans=max(ans,(leftmax[i-1]+rightmax[i+1]))

    return ans
```

```
list1=list(map(int,input().split(',')))
if max(list1)<=0:
    print(max(list1))
else :
    print(tuhaokadane(list1))
```

二分查找

```
# GPT-4o
def check():
    #……
    return

lo , hi = ? , ?
while lo < hi:
    mid = ( lo + hi )// 2
    if check(mid):
        ans = mid
        hi = mid
    else :
        lo = mid + 1

# 具体问题具体分析边界条件
```

*list*直接赋值的时候是赋了一个引用，`copy()`是浅拷贝

一维数组可以*copy*，多维数组需要*deepcopy*

```
from copy import deepcopy
list2=deepcopy(list1)
```

zip

当你有两个或更多个可迭代对象（例如列表、元组或其他序列）时，*zip*函数可以将它们逐个元素地配对。

它返回一个迭代器，该迭代器生成元组，每个元组包含来自每个可迭代对象的相应位置的元素。

```
list1 = [1, 2, 3]
list2 = ['a', 'b', 'c']

zipped = zip(list1, list2)

for pair in zipped:
    print(pair)

# 输出：
(1, 'a')
(2, 'b')
(3, 'c')
```

在代码中，`zip(list1, list2)`将*list*1和*list*2中的元素逐一配对，生成一个由元组组成的迭代器*zipped*。

然后，我们可以通过迭代*zipped*来访问这些配对。每次迭代，我们得到一个包含来自*list1*和*list2*对应位置的元素的元组。

需要注意的是，如果可迭代对象的长度不同，*zip*函数将会以最短的长度为准，多余的元素将被忽略。

heapq

默认是最小堆，但实际上把值都取负就是最大堆，*pop*的时候再取一下负就行了

```
# OJ-06648:Sequence
import heapq

def g(list2,list1,n):
    heap=[(list1[0]+list2[i],0,i)for i in range(n)]
    ans=[]
    while n>0:
        ans1,pos1,pos2=heapq.heappop(heap)
        ans.append(ans1)
        if pos1+1<n:
            heapq.heappush(heap,(list1[pos1+1]+list2[pos2],pos1+1,pos2))
        n-=1
    return sorted(ans)

def f(list1,n,m):
    mini=list1[0]
    for i in range(1,m):
        mini=g(mini,list1[i],n)
    return mini

for _ in range(int(input())):
    m,n=map(int,input().split())
    list1=[sorted(list(map(int,input().split())))for _ in range(m)]
    result=f(list1,n,m)
    print(*result)
```

deque

```
# OJ-26978:滑动窗口最大值
from collections import deque

n, k = map(int, input().split())
list1 = list(map(int, input().split()))
deque = deque()  # 用于存储索引
ans = []

for i in range(n):
    # 移除所有在当前滑动窗口之外的索引
    while deque and deque[0] < i - k + 1:
        deque.popleft()
    # 移除所有小于当前元素的值，因为它们不会成为最大值
    while deque and list1[i] >= list1[deque[-1]]:
        deque.pop()
    deque.append(i)
    # 当前窗口形成后，添加当前窗口的最大值到结果中
    if i >= k - 1:
        ans.append(list1[deque[0]])

print(*ans)

#我偏好的写法（虽然劣）：
n,k=map(int,input().split())
list1=list(map(int,input().split()))
```

```
import heapq
heap=[(-list1[0],0)]
ans=[]
pos=1
while pos<=n:
    val,pos1=heapq.heappop(heap)
    while pos-pos1>k and heap:
        val,pos1=heapq.heappop(heap)
    if pos>=k:
        ans.append(-val)
    heapq.heappush(heap,(val,pos1))
    if pos==n:
        break
    heapq.heappush(heap,(-list1[pos],pos))
    pos+=1

print(*ans)
```

前缀和

```
# OJ-20453:和为k的子数组个数
list1=list(map(int,input().split()))
k=int(input())

pre=[list1[0]]
for i in range(1,len(list1)):
    pre.append(pre[-1]+list1[i])

ans=pre.count(k)
for i in range(len(list1)):
    ans+=pre[:i].count(pre[i]-k)

print(ans)
```

`dict.get(value,type)`是$defaultlist$的很好替代

异或与(XOR)

XOR：一真一假为$True$，对数值进行XOR运算时实际上是在2进制下做运算，符号为`^`，有意思的是`a^a=0`

```
# OJ-20626:对子数列做XOR运算
list1=list(map(int,input().split()))
sum1=list1[0]
stack=[sum1]
for i in range(1,len(list1)):
    sum1^=list1[i]
    stack.append(sum1)

for _ in range(10000):
    a,b=map(int,input().split())
    if a==0:
        print(stack[b])
    else :
        print(stack[b]^stack[a-1])
# 对前缀和很好的应用
```

DP替代线段树

```
# OJ-23568:幸福的寒假生活
def change(a):
    mon,day=map(int,a.split('.'))
    return 31*(mon-1)+day-6

n=int(input())
list1=[]
dp=[0 for _ in range(46)]
for _ in range(n):
    a,b,c=input().split()
    a,b,c=change(a),change(b),int(c)
    if b>45:
        continue
    list1.append((a,b,c))
n=len(list1)
for i in range(1,46):
    dp[i]=dp[i-1]
    for j in list1:
        st,en,va=j[0],j[1],j[2]
        if en==i:
            dp[i]=max(dp[i],dp[st-1]+va)
print(dp[45])
```

AVL

```
# GPT-4o
class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
        self.height = 1

def get_height(root):
    if not root:
        return 0
    return root.height

def update_height(root):
    root.height = 1 + max(get_height(root.left), get_height(root.right))

def get_balance(root):
    if not root:
        return 0
    return get_height(root.left) - get_height(root.right)

def LL_rotate(root):
    a = root.left
    root.left = a.right
    a.right = root
    update_height(root)
    update_height(a)
    return a

def RR_rotate(root):
    a = root.right
    root.right = a.left
    a.left = root
```

```
    update_height(root)
    update_height(a)
    return a

def LR_rotate(root):
    root.left = RR_rotate(root.left)
    return LL_rotate(root)

def RL_rotate(root):
    root.right = LL_rotate(root.right)
    return RR_rotate(root)

class AVL:
    def __init__(self):
        self.root = None

    def insert(self, data):
        self.root = self._insert(self.root, data)

    def _insert(self, root, data):
        if not root:
            return Node(data)
        elif data < root.data:
            root.left = self._insert(root.left, data)
        else:
            root.right = self._insert(root.right, data)

        update_height(root)
        balance = get_balance(root)

        if balance > 1:
            if data < root.left.data:  # LL
                root = LL_rotate(root)
            else:  # LR
                root = LR_rotate(root)
        elif balance < -1:
            if data > root.right.data:  # RR
                root = RR_rotate(root)
            else:  # RL
                root = RL_rotate(root)

        return root
```

骑士周游

记得启发式搜索,其他开摆

KMP

```
# 前缀数组
def findpre(s):
    pre=[0 for _ in range(len(s))]
    j=0
    for i in range(1,len(s)):
        while j>0 and s[i]!=s[j]:
            j=pre[j-1]
        if s[i]==s[j]:
            j+=1
        pre[i]=j
    return pre
```

```
# next数组
next=[-1]+pre[:-1]
```

值得注意的是：

```
pos%(pos - pre[pos-1]) == 0 and pre[pos-1]!=0
```

是判定字符串前缀是否含有周期的条件

中序表达式转后序——Shunting Yard Algorithm

```
# GPT-4o
ops_to_num = {'+': 1, '-': 1, '*': 2, '/': 2}

def shunting_yard(expression):
    output = []
    ops = []
    for char in expression:
        if char.isdigit() or char == '.':
            output.append(char)
        elif char == '(':
            ops.append(char)
        elif char == ')':
            while ops and ops[-1] != '(':
                output.append(ops.pop())
            ops.pop()
        else:
            while ops and ops[-1] != '(' and ops_to_num[ops[-1]] >= ops_to_num[char]:
                output.append(ops.pop())
            ops.append(char)
    while ops:
        output.append(ops.pop())
    return ''.join(output)
```

Merge_Sort

```
# GPT-4o
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
    return merge(left, right)

def merge(left, right):
    merged = []
    left_idx, right_idx = 0, 0
    while left_idx < len(left) and right_idx < len(right):
        if left[left_idx] <= right[right_idx]:
            merged.append(left[left_idx])
            left_idx += 1
        else:
            merged.append(right[right_idx])
            right_idx += 1
    merged.extend(left[left_idx:])
    merged.extend(right[right_idx:])
    return merged
```

食物链

```
# OJ-01182
def findfather(x,father):
    if father[x]!=x:
        father[x]=findfather(father[x],father)
    return father[x]

def union(n,statement):
    father=[x for x in range(3*n+1)]
    ans=0
    for type,x,y in statement:
        if x>n or y>n or type==2 and x==y:
            ans+=1
            continue
        if findfather(x,father)==findfather(y,father) and type==2:
            ans+=1
        elif findfather(x,father)==findfather(y+n,father) and type==1:
            ans+=1
        elif findfather(y,father)==findfather(x+n,father) :
            ans+=1
        else:
            if type==1:
                father[findfather(y,father)]=findfather(x,father)
                father[findfather(y+n,father)]=findfather(x+n,father)
                father[findfather(y+2*n,father)]=findfather(x+2*n,father)
            else:
                father[findfather(y+n,father)]=findfather(x,father)
                father[findfather(y+2*n,father)]=findfather(x+n,father)
                father[findfather(y,father)]=findfather(x+2*n,father)

    print(ans)

n,k=map(int,input().split())
list1=[list(map(int,input().split())) for _ in range(k)]
union(n,list1)

# 如果a和b是同类，肯定有a_n与b_n连通，即有相同的根节点；如果a吃b，肯定有a_n与b_2n连通
```

Dijkstra

```
# CV来的
def dijkstra(start,end):
    heap=[(0,start,[start])]
    vis=set()
    while heap:
        (cost,u,path)=heappop(heap)
        if u in vis: continue
        vis.add(u)
        if u==end: return (cost,path)
        for v in graph[u]:
            if v not in vis:
                heappush(heap,(cost+graph[u][v],v,path+[v]))
```

kruskal—对边进行操作，稠密图劣于*Prim*

```
学不会，不想学
```

prim—用*heap*就行

拓扑排序—每次找入度为0的点,可以用来查找图中是否有环

工具

```
for key,value in dict.items() # 遍历字典的键值对。
for index,value in enumerate(list) # 枚举列表，提供元素及其索引。
math.pow(m,n) # 计算 m 的 n 次幂。
math.log(m,n) # 计算以 n 为底的 m 的对数。

from functools import lru_cache
@lru_cache(maxsize=None)
```

字符串

```
#返回子字符串sub在字符串中首次出现的索引，如果未找到，则返回-1。
str.find(sub)

#将字符串中的old子字符串替换为new(不超过max次，不加就全部替换)。
str.replace(old, new [,max])

#检查字符串是否以prefix开头或以suffix结尾。
str.startswith(prefix)
str.endswith(suffix)

#检查字符串是否全部由字母/数字/字母和数字组成。
str.isalpha()
str.isdigit()
str.isalnum()
```

permutations：全排列

```
from itertools import permutations
perm = permutations([1, 2, 3])
print(*perm)
# 输出: (1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)
```

combinations：组合

```
from itertools import combinations
comb = combinations([1, 2, 3], 2)
print(*comb)
# 输出: (1, 2), (1, 3), (2, 3)
```

字符串大小写转换

```
text.upper() # 变全大写
text.lower() # 变全小写
text.capitalize() # 首字母大写
text.swapcase() # 大小写转换
```

`str.find()`和`str.index()`

```
my_string = "Hello, world!"
index = my_string.find("world")
print(index)  # 输出：7

index = my_string.find("Python")
print(index)  # 输出：-1

my_string = "Hello, world!"
index = my_string.index("world")
print(index)  # 输出：7

index = my_string.index("Python")  # 引发 ValueError
```

集合

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}

# 并集
union_set = set1 | set2 # {1, 2, 3, 4, 5}

# 交集
intersection_set = set1 & set2 # {3}

# 差集
difference_set = set1 - set2 # {1, 2}

# 对称差集
symmetric_difference_set = set1 ^ set2 # {1, 2, 4, 5}
```

import

```
import heapq
from collections import defaultdict
import bisect
from bisect import *
from functools import lru_cache
@lru_cache(maxsize=None)
import math
math.ceil()  # 函数进行向上取整
math.floor() # 函数进行向下取整。
round()      # 四舍五入
```

bisect

1. `bisect.bisect_left(list, x, lo=0, hi=len(list))`
   - 在列表`list`中查找元素`x`的插入点，使得插入后仍保持排序。
   - 返回插入点的索引，插入点位于`list`中所有等于`x`的元素之前。
2. `bisect.bisect_right(list, x, lo=0, hi=len(list))`
   - 类似于`bisect_left`，但插入点位于`list`中所有等于`x`的元素之后。
3. `bisect.insort_left(list, x, lo=0, hi=len(list))`
   - 在`list`中查找`x`的插入点并插入`x`，保持列表`list`的有序。
   - 插入点位于`list`中所有等于`x`的元素之前。
4. `bisect.insort_right(list, x, lo=0, hi=len(list))`
   - 类似于`insort_left`，但插入点位于`list`中所有等于`x`的元素之后。

```
a = [1, 2, 4, 4, 5]

# 查找插入点
print(bisect.bisect_left(a, 4))  # 输出: 2
print(bisect.bisect_right(a, 4)) # 输出: 4

# 插入元素
bisect.insort_left(a, 3)
print(a)  # 输出: [1, 2, 3, 4, 4, 5]

bisect.insort_right(a, 4)
print(a)  # 输出: [1, 2, 3, 4, 4, 4, 5]
```

$nlog(n)$复杂度求逆序数

```
import bisect
n=int(input())
lsit1=list(map(int,input().split()))
list1=[]
ans=0
for i in lsit1:
    pos=bisect.bisect_left(list1,i)
    ans+=pos
    list1.insert(pos,i)
print(ans)
```

进制转换

```
b = bin(item)  # 2进制 0b1111
o = oct(item)  # 8进制 0o1111
h = hex(item)  # 16进制 0xffff
int(str,n)     #将字符串str转换为n进制的整数。
int(bin(item)[2:])   #十进制转二进制
#任意阶：辗转相除
def f(n,x):
    a=[0,1,2,3,4,5,6,7,8,9,'A','b','C','D','E','F']#……
    b=[]
    while n:
        b.append(a[n%x])
        n//=x
    return (b[::-1])
```

$ASCII$

```
ord(char) -> ASCII_value
chr(ascii_value) -> char
```

$Trie$

```
#自己写的破烂 OJ-04089
class Node:
    def __init__(self,data=None):
        self.data=data
        self.children={}

class Trie:
    def __init__(self,list2):
        self.list2=list2
        self.root=Node()

    def buildtree(self):
        for i in self.list2:
            a=self.root
            for char in i:
                if char not in a.children.keys():
                    a.children[char]=Node(char)
                    a=a.children[char]
                else:
                    a=a.children[char]

    def search(self):
        for i in self.list2:
            a=self.root
            for pos in range(len(i)):
                a=a.children[i[pos]]
            if a.children:
                return False
        return True

for _ in range(int(input())):
    n=int(input())
    list1=[input() for _ in range(n)]
    if len(list1)!=len(set(list1)):
        print('NO')
        continue
    Tree=Trie(list1)
    Tree.buildtree()
    if Tree.search():
        print('YES')
    else:
        print('NO')
```

2.强联通子图

Kosaraju's算法可以分为以下几个步骤：

1. 第一次DFS：对图进行一次DFS，并记录每个顶点的完成时间（即DFS从该顶点返回的时间）。
2. 转置图：将图中所有边的方向反转，得到转置图。
3. 第二次DFS：根据第一次DFS记录的完成时间的逆序，对转置图进行DFS。每次DFS遍历到的所有顶点构成一个强连通分量。

详细步骤

1. 第一次DFS:
   - 初始化一个栈用于记录DFS完成时间顺序。
   - 对图中的每个顶点执行DFS，如果顶点尚未被访问过，则从该顶点开始DFS。
   - DFS过程中，当一个顶点的所有邻居都被访问过后，将该顶点压入栈中。
2. 转置图:
   - 创建一个新的图，边的方向与原图相反。
3. 第二次DFS:
   - 初始化一个新的访问标记数组。
   - 根据栈中的顺序（即第一步中记录的完成时间的逆序）对转置图进行DFS。
   - 每次从栈中弹出一个顶点，如果该顶点尚未被访问过，则从该顶点开始DFS，每次DFS遍历到的所有顶点构成一个强连通分量。

示例代码

以下是Kosaraju's算法的Python实现:

```
from collections import defaultdict

class Graph:
    def __init__(self, vertices):
        self.graph = defaultdict(list)
        self.V = vertices

    def addEdge(self, u, v):
        self.graph[u].append(v)

    def _dfs(self, v, visited, stack):
        visited[v] = True
        for neighbour in self.graph[v]:
            if not visited[neighbour]:
                self._dfs(neighbour, visited, stack)
        stack.append(v)

    def _transpose(self):
        g = Graph(self.V)
        for i in self.graph:
            for j in self.graph[i]:
                g.addEdge(j, i)
        return g

    def _fillOrder(self, v, visited, stack):
        visited[v] = True
        for neighbour in self.graph[v]:
            if not visited[neighbour]:
                self._fillOrder(neighbour, visited, stack)
        stack.append(v)

    def _dfsUtil(self, v, visited):
        visited[v] = True
        print(v, end=' ')
        for neighbour in self.graph[v]:
            if not visited[neighbour]:
                self._dfsUtil(neighbour, visited)

    def printSCCs(self):
        stack = []
        visited = [False] * self.V

        for i in range(self.V):
            if not visited[i]:
```

```
                self._fillOrder(i, visited, stack)

        gr = self._transpose()

        visited = [False] * self.V

        while stack:
            i = stack.pop()
            if not visited[i]:
                gr._dfsUtil(i, visited)
                print("")

# 示例使用
g = Graph(5)
g.addEdge(1, 0)
g.addEdge(0, 2)
g.addEdge(2, 1)
g.addEdge(0, 3)
g.addEdge(3, 4)

print("Strongly Connected Components:")
g.printSCCs()
```