



Universidad de Sevilla

Teorema del Sándwich de Ham

Kenny Flores, Pablo Dávila, Pablo Reina

2025-03-10

Índice

1. Setup	2
2. Estructuras de Datos	2
3. Algoritmos de Búsqueda y ordenamiento	4
4. Grafos	4
5. Matemáticas	5
6. Strings	6
7. Geometría	6

1. Setup

Al trabajar en la terminal de Linux, existen tres métodos principales para ejecutar los programas asociados a los problemas. Estos métodos facilitan la interacción con los archivos de entrada y salida, optimizando el flujo de trabajo. A continuación, se describen estas opciones y se ejemplifican en detalle.

1.1. Lectura de números enteros

Ejemplo: El programa .py recibe un número seguido de una lista de enteros y devuelve la suma de todos los elementos de la lista.

- (Opción 1) Escribir los valores de entrada a mano, esto ejecuta el programa y escribe los datos directamente en la terminal.

```
$ python3 problema.py
# Ejemplo:
# Input:
# 3
# 10 20 30
# Output:
# 60
```

- (Opción 2) Hacer uso de una tubería, esto pasa el contenido de un archivo como entrada al programa utilizando una tubería.

```
$ cat input.txt | python3 problema.py
# Ejemplo:
# input.txt contiene:
# 3
# 10 20 30
```

```
# Output:
# 60
```

- (Opción 3) Ejecutar el programa con entrada desde un archivo y guardar la salida en otro archivo

```
$ python3 problema.py < input.txt > output.txt
# Ejemplo:
# input.txt contiene:
# 3
# 10 20 30
# Después de ejecutar:
# output.txt contendrá:
# 60
```

1.2. Ajuste del Limite de Recursión

En problemas de programación competitiva, es común trabajar con algoritmos recursivos. Sin embargo, Python establece un límite predeterminado en la profundidad de recursión (por defecto 1000) para evitar desbordamientos de pila (stack overflow).

Este límite puede ser insuficiente en problemas que requieren recursión profunda. Para ajustar este límite, se utiliza la función `sys.setrecursionlimit`. Por ejemplo:

```
import sys
sys.setrecursionlimit(300001)
```

Consideraciones importantes

- **Uso Responsable:** Aumentar el límite de recursión puede causar un mayor consumo de memoria y potencialmente llevar a errores si la pila del sistema no puede manejar la carga.
- **Problemas Comunes:** Si el programa lanza un error del tipo `RecursionError: maximum recursion depth exceeded`, esto indica que se alcanzó el límite actual.
- **Límite Recomendado:** En la mayoría de los problemas competitivos, un valor de `10**6` suele ser suficiente, pero esto depende del problema y del entorno en el que se ejecute

1.3. Optimización de entrada

Reemplazamos por `.stdin.readline` para mejorar la velocidad de lectura, especialmente en ciclos con grandes volúmenes de datos.

```
import sys
input = sys.stdin.readline
```

#Ejemplo de uso:

```
# Leer una línea de entrada y eliminar el salto de línea
n = int(input().strip()) # Usamos strip() para eliminar el salto de línea adicional
```

Consideraciones:

- `.stdin.readline()` lee la línea completa, por lo que se usa `strip()` para quitar el salto de línea.
- `.stdin.readline()` devuelve una cadena, por lo que hay que convertirla a otros tipos manualmente.

2. Estructuras de Datos

2.1. Listas

Las listas en Python son dinámicas y versátiles. Son ideales para almacenar secuencias de elementos.

Operaciones comunes

```
# Crear una lista
arr = [1, 2, 3, 4, 5]
```

```
# Acceder a un elemento O(1)
print(arr[2]) # Output: 3
```

```
# Añadir un elemento al final O(1)
arr.append(6) # arr = [1, 2, 3, 4, 5, 6]
```

```
# Eliminar un elemento O(n)
arr.pop() # arr = [1, 2, 3, 4, 5]
arr.pop(2) # arr = [1, 2, 4, 5]
```

```
# Búsqueda O(n)
if 4 in arr:
    print("4 está en la lista")
```

2.2. Pilas (Stacks)

Una pila es una estructura LIFO (Last In, First Out). Se puede implementar con una lista.

Operaciones comunes:

```
stack = []
```

```
# Push: añadir elemento en la cima de la pila O(1)
stack.append(1) # stack = [1]
stack.append(2) # stack = [1, 2]
```

```
# Pop: Borra y devuelve elemento que está en la cima de la pila O(1)
top = stack.pop() # top = 2, stack = [1]
```

```
# Top: ver el tope O(1)
if stack:
    print(stack[-1]) # Output: 1
```

2.3. Colas (Queues)

Una cola es una estructura FIFO (First In, First Out). Para eficiencia, usa deque de la librería .

```
from collections import deque
```

```
queue = deque()
```

```
# Enqueue: Añade el elemento al final de la cola O(1)
queue.append(1) # queue = [1]
queue.append(2) # queue = [1, 2]
```

```
# Dequeue: Elimina y devuelve el primer elemento de la cola O(1)
front = queue.popleft() # front = 1, queue = [2]
```

```
# Ver el frente
if queue:
    print(queue[0]) # Output: 2
```

2.4. Sets

Almacenan los elementos únicos y permiten operaciones de búsqueda e inserción eficientes

```
s = set()
```

```
# Añadir elementos O(1)
s.add(1) # s = {1}
s.add(2) # s = {1, 2}

# Eliminar elementos O(1)
s.remove(1) # s = {2}

# Búsqueda O(1)
if 2 in s:
    print("2 está en el conjunto")
```

2.5. Diccionarios

Los diccionarios almacenan pares clave-valor. Son ideales para búsquedas rápidas.

```
d = {}

# Añadir elementos O(1)
d["a"] = 1 # d = {"a": 1}
d["b"] = 2 # d = {"a": 1, "b": 2}
```

```
# Acceder a un valor O(1)
print(d["a"]) # Output: 1
```

```
# Eliminar un elemento O(1)
del d["a"] # d = {"b": 2}
```

```
# Búsqueda O(1)
if "b" in d:
    print("b está en el diccionario")
```

2.6. Heap (Colas de prioridad)

Un heap es una estructura que permite obtener el elemento mínimo o máximo eficientemente. En Python, se usa la librería .

Operaciones comunes:

- Push: $O(\log n)$
- Pop: $O(\log n)$
- Top: $O(1)$

```
import heapq
```

```
# Crear un min-heap
heap = []
```

```
# Realizar un push  $O(\log n)$ 
```

```
heapq.heappush(heap, 3) # heap = [3]
heapq.heappush(heap, 1) # heap = [1, 3]
heapq.heappush(heap, 2) # heap = [1, 3, 2]
```

```
# Obtener el elemento mínimo
print(heap[0]) # Output: 1
```

```
# Eliminar el elemento mínimo
top = heapq.heappop(heap) # top = 1, heap = [2, 3]
```

2.7. Cola Doble (Deque)

Un deque permite operaciones eficientes en ambos extremos. Es útil para problemas de ventanas deslizantes o BFS.

Operaciones comunes:

```
from collections import deque
```

```
dq = deque()
```

```
# Añadir elementos O(1)
dq.append(1) # dq = [1]
dq.appendleft(2) # dq = [2, 1]
```

```
# Eliminar elementos O(1)
front = dq.popleft() # front = 2, dq = [1]
back = dq.pop() # back = 1, dq = []
```

2.8. Counter

Counter de la librería es útil para contar frecuencias de elementos en una lista.

Operaciones comunes:

```
from collections import Counter
```

```
arr = [1, 2, 2, 3, 3, 3]
# Conteo  $O(n)$ 
counter = Counter(arr)
```

```
print(counter) # Output: Counter({3: 3, 2: 2, 1: 1})
```

```
# Acceder a la frecuencia de un elemento  $O(1)$ 
print(counter[2]) # Output: 2
```

2.9. Segment Tree

Es una estructura empleada para optimizar operaciones sobre rangos (segmentos) de un array.

```
class Node(object):
    def __init__(self, start, end):
        self.start = start
        self.end = end
        self.total = 0
        self.left = None
        self.right = None

class SegmentTree(object):
    def __init__(self, arr):
        def createTree(arr, l, r):
            if l > r: # Base case
                return None
            if l == r: # Leaf
                n = Node(l, r)
                n.total = arr[l]
                return n
            mid = (l + r) // 2
            root = Node(l, r)
            # A tree is a recursive structure
            root.left = createTree(arr, l, mid)
            root.right = createTree(arr, mid + 1, r)
            # This is the part we change between
            trees
            root.total = root.left.total +
            root.right.total
            return root

        self.root = createTree(arr, 0, len(arr) - 1)

    def update(self, i, val):
        def updateVal(root, i, val):
            # Base case. The actual value will be
            updated in a leaf.
            if root.start == root.end:
                root.total = val
                return val
            mid = (root.start + root.end) // 2
            # If the index is less than mid, that
            leaf must be in the left segment
            if i <= mid:
                updateVal(root.left, i, val)
            # Otherwise, the right segment
```

```
        else:
            updateVal(root.right, i, val)
        # Propagate upwards
        root.total = root.left.total +
        root.right.total
        return root.total
    return updateVal(self.root, i, val)

    def sumRange(self, i, j):
        # Helper function to calculate range sum
        def rangeSum(root, i, j):
            # If the range exactly matches the root,
            we already have the sum
            if root.start == i and root.end == j:
                return root.total
            mid = (root.start + root.end) // 2
            if j <= mid:
                return rangeSum(root.left, i, j)
            elif i >= mid + 1:
                return rangeSum(root.right, i, j)
            else:
                return rangeSum(root.left, i, mid) +
                rangeSum( root.right, mid + 1, j )

        return rangeSum(self.root, i, j)
```

3. Algoritmos de Búsqueda y ordenamiento

3.1. Búsqueda binaria

Busca en una lista ordenada dividiéndola a la mitad en cada paso. La complejidad en tiempo es de $O(\log n)$.

```
# Modificar según lo que se necesite
def binary_search(arr, target):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1
```

```
# Ejemplo de uso
arr = [1, 3, 5, 7, 9]
print(binary_search(arr, 5)) # Output: 2
```

3.2. Algoritmo sorted de Python

El método sorted en Python es una función incorporada que ordena cualquier iterable (listas, tuplas, etc.) y devuelve una nueva lista ordenada. Internamente, utiliza una combinación de algoritmos eficientes, como Timsort (un híbrido de Merge Sort e Insertion Sort), que garantiza un rendimiento óptimo en la mayoría de los casos. La complejidad en tiempo en el peor de los casos es $O(n \log n)$.

```
# Lista de ejemplo
arr = [3, 1, 4, 1, 5, 9, 2, 6, 5]

# Uso básico de sorted
sorted_arr = sorted(arr) # Orden ascendente
print("Orden ascendente:", sorted_arr)
```

```
# Orden descendente
sorted_arr_desc = sorted(arr, reverse=True)
print("Orden descendente:", sorted_arr_desc)
```

```
# Ordenamiento personalizado con key
words = ["apple", "bat", "carrot", "dog"]
sorted_words = sorted(words, key=len) # Ordenar por longitud
print("Ordenado por longitud:", sorted_words)
```

```
# Ordenamiento de tuplas por un elemento específico
people = [("Alice", 25), ("Bob", 20), ("Charlie", 30)]
sorted_people = sorted(people, key=lambda x: x[1])
# Ordenar por edad
print("Ordenado por edad:", sorted_people)
```

4. Grafos

4.1. Clase Grafos

```
class Graph:
    def __init__(self):
        self.neighbors = []
        self.name2node = {}
        self.node2name = []
        self.weight = []
```

```

def __len__(self):
    return len(self.name2name)

def __getitem__(self, v):
    return self.neighbors[v]

def add_node(self, name):
    assert name not in self.name2node
    self.name2node[name] = len(self.name2node)
    self.name2name.append(name)
    self.neighbors.append([])
    self.weight.append({})
    return self.name2node[name]

def add_edge(self, name_u, name_v,
weight_uv=None):
    self.add_arc(name_u, name_v, weight_uv)
    self.add_arc(name_v, name_u, weight_uv)

def add_arc(self, name_u, name_v,
weight_uv=None):
    u = self.name2node[name_u]
    v = self.name2node[name_v]
    self.neighbors[u].append(v)
    self.weight[u][v] = weight_uv

```

4.2. Búsqueda en Profundidad (DFS)

Recorre todos los nodos de un grafo o árbol en profundidad.

```

def dfs_iterative(graph, start, seen):
    seen[start] = True
    to_visit = [start]
    while to_visit:
        node = to_visit.pop()
        for neighbor in graph[node]:
            if not seen[neighbor]:
                seen[neighbor] = True
                to_visit.append(neighbor)

```

4.3. Búsqueda en Amplitud (BFS)

Recorre todos los nodos de un grafo o árbol nivel por nivel.

```

from collections import deque

def bfs(graph, start=0):
    to_visit = deque()

```

```

dist = [float('inf')] * len(graph)
prec = [None] * len(graph)
dist[start] = 0
to_visit.appendleft(start)

while to_visit:
    node = to_visit.pop()
    for neighbor in graph[node]:
        if dist[neighbor] == float('inf'):
            dist[neighbor] = dist[node] + 1
            prec[neighbor] = node
            to_visit.appendleft(neighbor)

return dist, prec

```

4.4. Flujo máximo

```

from collections import deque

def bfs(capacity, source, sink, parent):
    queue = deque([source])
    visited = set([source])
    while queue:
        u = queue.popleft()
        for v in range(len(capacity)):
            if v not in visited and capacity[u][v] > 0:
                queue.append(v)
                visited.add(v)
                parent[v] = u
                if v == sink:
                    return True
    return False

def edmonds_karp(capacity, source, sink):
    max_flow = 0
    n = len(capacity)
    parent = [-1] * n

    while bfs(capacity, source, sink, parent):
        path_flow = float('inf')
        v = sink
        while v != source:
            u = parent[v]
            path_flow = min(path_flow, capacity[u][v])
            v = u

```

```

v = sink
while v != source:
    u = parent[v]
    capacity[u][v] -= path_flow
    capacity[v][u] += path_flow
    v = u

max_flow += path_flow

return max_flow

```

5. Matemáticas

5.1. Algoritmo de Euclides (GCD)

Algoritmo para calcular el máximo común divisor

```

def gcd(a, b):
    return a if b==0 else gcd(b,a%b)

```

5.2. Mínimo común múltiplo (LCM)

```

def lcm(a, b):
    return abs(a) // gcd(a, b) * abs(b)

```

5.3. Criba de Eratóstenes

Algoritmo para generar números primos

```

def eratosthene(limit):
    primes = [2, 3, 5]
    sieve = [True] * (limit + 1)
    sieve[0] = sieve[1] = False

    for i in range(2, int(limit**0.5) + 1):
        if sieve[i]:
            for j in range(i * i, limit + 1, i):
                sieve[j] = False

    return [i for i in range(2, limit + 1) if sieve[i]]

```

5.4. Exponenciación rápida

Calcula $a^b \bmod q$ de manera eficiente $O(\log b)$.

```

def fast_exponentiation(a,b,q):
    assert a >= 0 and b>=0 and q>=1
    result = 1
    while b:
        if b%2==1:

```

```

    result = (result * a) % q
    a=(a*a)%q
    b>>=1
return result

```

5.5. Coeficientes binomiales

El coeficiente binomial $\binom{n}{k}$ representa el número de formas de elegir un subconjunto de k elementos de un conjunto de n elementos. Por ejemplo $\binom{4}{2} = 6$ porque el conjunto $\{1, 2, 3, 4\}$ tiene 6 subconjuntos de 2 elementos $\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 3\}, \{2, 4\}, \{3, 4\}$. Una de las formas más eficientes de calcular estos coeficientes es mediante la siguiente fórmula $\binom{n}{k} = \frac{n!}{k!(n-k)!}$

```
MAXN = 10**6
```

```

fac = [0] * (MAXN + 1)
inv = [0] * (MAXN + 1)

```

```

def exp(x: int, n: int, m: int) -> int:
    """return: x^n modulo m in O(log p) time."""
    x %= m # note: m * m must be less than 2^63 to
avoid ll overflow
    res = 1
    while n > 0:
        if n % 2 == 1:
            res = (res * x) % m
        x = (x * x) % m
        n //= 2
    return res

```

```

def factorial(p: int):
    """Precomputes n! from 0 to MAXN."""
    global fac
    fac[0] = 1
    for i in range(1, MAXN + 1):
        fac[i] = (fac[i - 1] * i) % p

```

```

def inverses(p: int):
    """
    Precomputes all modular inverse factorials from 0
    to MAXN in O(n + log p) time
    """

```

```

global inv
inv[MAXN] = exp(fac[MAXN], p - 2, p)
for i in range(MAXN, 0, -1):
    inv[i - 1] = (inv[i] * i) % p

```

```

def choose(n: int, r: int, p: int):
    """return: nCr mod p"""
    return fac[n] * inv[r] % p * inv[n - r] % p

```

6. Strings

6.1. Trie

```

class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end_of_word = False

class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        node.is_end_of_word = True

    def search(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                return False
            node = node.children[char]
        return node.is_end_of_word

    def starts_with(self, prefix):
        node = self.root
        for char in prefix:
            if char not in node.children:
                return False
            node = node.children[char]
        return True

```

7. Geometría

7.1. Clase Punto

```

import math

class Pt:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    # Sobrecarga de operadores
    def __add__(self, other):
        return Pt(self.x + other.x, self.y +
other.y)

    def __mul__(self, k):
        return Pt(k * self.x, k * self.y)

    def __neg__(self):
        return Pt(-self.x, -self.y)

    def __sub__(self, other):
        return self + (-other)

    def __truediv__(self, d):
        if d == 0:
            raise ValueError("operator/ Recuerdos de
vietADAMan (p/0)")
        return self * (1 / d)

    def __iadd__(self, other):
        self.x += other.x
        self.y += other.y
        return self

    def __isub__(self, other):
        self.x -= other.x
        self.y -= other.y
        return self

    def __imul__(self, k):
        self.x *= k
        self.y *= k
        return self

    def __itruediv__(self, k):
        self.x /= k

```

```

        self.y /= k
        return self

    def __eq__(self, other):
        return self.x == other.x and self.y ==
other.y

    def __ne__(self, other):
        return not (self == other)

    def __gt__(self, other):
        return (self.x == other.x and self.y >
other.y) or (self.x > other.x)

    def __lt__(self, other):
        return other > self

    def rot(self):
        return Pt(-self.y, self.x) # Rotar 90
grados en sentido antihorario

    def __repr__(self):
        return f"{self.x} {self.y}"

# Funciones adicionales
def dot(p, q):
    return p.x * q.x + p.y * q.y

def norma(p):
    return math.sqrt(dot(p, p))

def angle(p, q):
    return math.acos(dot(p, q) / (norma(p) *
norma(q)))

def cross(p, q):
    return p.x * q.y - p.y * q.x

def collinear(a, b):
    return cross(a, b) == 0 # Ver si dos puntos son
colineales

def orient(a, b, c):
    return cross(b - a, c - a)

def orientS(a, b, c):
    return (orient(a, b, c) > 0) - (orient(a, b, c)
< 0) # Signo de la orientación

```

```

def cw(a, b, c, in_colli=False):
    o = orientS(a, b, c)
    return o < 0 or (in_colli and o == 0) # Sentido
horario

def ccw(a, b, c, in_colli=False):
    o = orientS(a, b, c)
    return o > 0 or (in_colli and o == 0) # Sentido
antihorario

def collinear_three(a, b, c):
    return orient(a, b, c) == 0 # Ver si tres
puntos están alineados

```

7.2. Clase Línea

```

import math

class Line:
    def __init__(self, *args):
        if len(args) == 1: # Constructor con un
solo argumento (vector)
            self.p = Pt(0, 0) # Punto de referencia
(origen)
            self.v = args[0] # Vector director
        elif len(args) == 2: # Constructor con dos
argumentos (punto y vector)
            self.p = args[0] # Punto de referencia
            self.v = args[1] # Vector director
        elif len(args) == 3: # Constructor con tres
argumentos (ax + by + c = 0)
            a, b, c = args
            if b == 0: # Si b es 0, la línea es
vertical
                self.p = Pt(-c / a, 0)
            else: # Si no, la línea pasa por (0, -
c/b)
                self.p = Pt(0, -c / b)
            self.v = Pt(b, -a) # Vector director

    def in_line(self, q):
        return collinear(self.p - q, self.v) # Ver
si q está en la línea

    def __eq__(self, other):
        return self.in_line(other.p) and
collinear(self.v, other.v) # Ver si dos líneas son

```

iguales

```

    def __ne__(self, other):
        return not (self == other) # Ver si dos
líneas son diferentes

    def perp(self, b):
        return Line(b, self.v.rot()) # Línea
perpendicular que pasa por b

    def __repr__(self):
        return f"Line(p={self.p}, v={self.v})"

# Funciones adicionales
def intersection(l1, l2):
    if l1 == l2:
        raise ValueError("intersection() son la
misma línea") # Líneas iguales
    d = cross(l1.v, l2.v) # Determinante
    c1 = cross(l1.v, l1.p)
    c2 = cross(l2.v, l2.p)
    if d == 0: # Líneas paralelas
        return []
    return [(l2.v * c1 - l1.v * c2) / d] # Punto de
intersección

def proj(p, r):
    return intersection(r, r.perp(p))[0] #
Proyección de p sobre r

def dist(p, q):
    return norma(p - q) # Distancia entre dos
puntos

def dist_line_point(p, r):
    return dist(p, proj(p, r)) # Distancia de un
punto a una línea

def dist_line_line(r, s):
    if collinear(r.v, s.v): # Líneas paralelas
        return dist(r.p, s.p)
    return 0 # Líneas que se intersectan

# 7.3. Clase Polígono
class Poly:
    def __init__(self, pts=None):
        if pts is None:

```

```

        self.pts = []
    else:
        self.pts = pts

    def perim(self):
        result = 0
        n = len(self.pts)
        for i in range(n):
            result += dist(self.pts[i], self.pts[(i
+ 1) % n])
        return result

    def isConvex(self):
        hasPos = False
        hasNeg = False
        n = len(self.pts)
        for i in range(n):
            o = orientS(self.pts[i], self.pts[(i +
1) % n], self.pts[(i + 2) % n])
            if o > 0:
                hasPos = True
            if o < 0:
                hasNeg = True
        return not (hasPos and hasNeg)

    def area(self):
        result = 0
        n = len(self.pts)
        for i in range(n):
            result += cross(self.pts[i], self.pts[(i
+ 1) % n])
        return abs(result) / 2

    def inPol(self, point):
        if not self.pts:
            return False
        sum_angle = 0
        n = len(self.pts)
        for i in range(n):
            if point == self.pts[i] or point ==
self.pts[(i + 1) % n]:
                return True
            sum_angle += angle(self.pts[i] - point,
self.pts[(i + 1) % n] - point) * (
                2 * ccw(point, self.pts[i],
self.pts[(i + 1) % n], False) - 1
            )
        return eq(abs(sum_angle), 2 * math.pi)

```

```

    def inConvex(self, point, strict=True):
        a = 1
        b = len(self.pts) - 1
        if orientS(self.pts[0], self.pts[a],
self.pts[b]) > 0:
            a, b = b, a
        if (
            orientS(self.pts[0], self.pts[a], point)
> 0
            or orientS(self.pts[0], self.pts[b],
point) < 0
            or (strict and (orientS(self.pts[0],
self.pts[a], point) == 0 or orientS(self.pts[0],
self.pts[b], point) == 0))
        ):
            return False
        while abs(a - b) > 1:
            c = (a + b) // 2
            if orientS(self.pts[0], self.pts[c],
point) > 0:
                b = c
            else:
                a = c
        return orientS(self.pts[a], self.pts[b],
point) < 0 or (not strict)

    def convex_hull(self, in_colli=False):
        if not self.pts:
            return
        p0 = min(self.pts)

        # Función de comparación para ordenar los
puntos
        def compare(a, b):
            o = orientS(p0, a, b)
            if o < 0:
                return -1
            elif o > 0:
                return 1
            else:
                return -1 if dist(p0, a) < dist(p0,
b) else 1

        # Ordenar los puntos usando la función de
comparación
        self.pts.sort(key=cmp_to_key(compare))

```

```

        if in_colli:
            i = len(self.pts) - 1
            while i > 0 and collinear(p0,
self.pts[i], self.pts[-1]):
                i -= 1
            self.pts = self.pts[:i + 1] + self.pts[i
+ 1][::-1]

        st = []
        for p in self.pts:
            while len(st) > 1 and not cw(st[-2],
st[-1], p, in_colli):
                st.pop()
            st.append(p)

        if not in_colli and len(st) == 2 and st[0]
== st[1]:
            st.pop()

        self.pts = st

        # Teorema de Pick
        def pick(I, B):
            return I - 1 + B * 0.5

```