Universidad de Sevilla

Teorema del Sándwich de Ham

Kenny Flores, Pablo Dávila, Pablo Reina

2025-08-28

Índice

1. Preparación concurso

template.py

```python
import sys

# Ajustar el límite de recursión
sys.setrecursionlimit(10**6)

"""
Descomenta para usar sys.stdin.readline en lugar de
input() y acelerar la lectura de datos. Advertencia:
sys.stdin.readline incluye un salto de línea (\n), por
lo que debes usar .strip() para eliminarlo.

Por ejemplo:
Leer una línea de entrada y eliminar el salto de línea
Usamos strip() para eliminar el salto de línea
n = int(input().strip())

"""
# input = sys.stdin.readline

def leer_un_numero():
    return int(input())

def leer_varios_numeros():
    return list(
        map(int, input().split())
    )

def main():
    # Código principal aquí
    pass

if __name__ == "__main__":

    main()
```

run.sh

```sh
# Usa este comando para compilar y ejecutar tu programa
# con entradas desde un archivo (ej: input.txt)
cat input.txt | python3 sol.py
```

troubleshoot.txt

```
Antes de enviar (Pre-submit)
- Escribe algunos casos de prueba simples si las muestras no son suficientes.
- ¿Los límites de tiempo están ajustados? Si es así, genera casos máximos.
- ¿El uso de memoria está bien?
- Asegúrate de enviar el archivo correcto.

Respuesta incorrecta (Wrong answer)
- ¡Usa print() para depurar tu solución! Imprime también la salida de depuración.
- ¿Estás limpiando todas las estructuras de datos entre casos de prueba? Asegúrate de reiniciar variables y estructuras como listas, diccionarios o conjuntos.
- ¿Tu algoritmo puede manejar todo el rango de entrada?
- Lee de nuevo el enunciado completo del problema.
- ¿Estás manejando correctamente todos los casos límite?
- ¿Has entendido bien el problema?
- ¿Alguna variable sin inicializar?
- ¿Confusión entre n y m, i y j, etc.?
- ¿Seguro de que tu algoritmo funciona?
- ¿Qué casos especiales no has considerado?
- ¿Seguro de que las funciones integradas o de las librerías que usas funcionan como piensas?
- Añade algunas aserciones (assert).
- Crea algunos casos de prueba para ejecutar tu algoritmo.
- Recorre el algoritmo con un caso sencillo.
- Vuelve a revisar esta lista.
- Explica tu algoritmo a un compañero.
- Pídele a tu compañero que revise tu código.
- Da un pequeño paseo, por ejemplo al baño.
- ¿El formato de salida es correcto? (incluyendo espacios en blanco)
- Reescribe tu solución desde cero o deja que un compañero lo haga.

Error en tiempo de ejecución (Runtime error)
- ¿Has probado todos los casos límite de forma local?
- ¿Alguna variable sin inicializar?
- ¿Estás accediendo a un índice fuera de rango en una lista, tupla o cadena?
- ¿Alguna aserción que pueda fallar?
- ¿Alguna posible división por cero?
- ¿Alguna recursión infinita posible?
- ¿Estás usando demasiada memoria? (Esto puede causar un MemoryError).

Límite de tiempo excedido (Time limit exceeded)
- ¿Tienes algún bucle infinito posible?
- ¿Cuál es la complejidad de tu algoritmo? Analiza si es O(N^2), O(N log N), etc.
- ¿Estás copiando muchos datos innecesarios?
- ¿Qué tan grande es la entrada y la salida? Considera usar sys.stdin.readline() para entradas grandes.
- ¿Estás usando estructuras de datos ineficientes para la operación que estás realizando? La búsqueda en una lista es O(N), mientras que en un conjunto o diccionario es O(1) en promedio.
- ¿Qué opinan tus compañeros sobre tu algoritmo?

Límite de memoria excedido (Memory limit exceeded)
- ¿Cuál es la cantidad máxima de memoria que debería necesitar tu algoritmo?
- ¿Estás limpiando todas las estructuras de datos entre casos de prueba?
- Evita crear listas o diccionarios innecesariamente grandes. Usa generadores o yield si es posible.
```

indefinite_read.py

```python
# Cuando no sabemos cuántas líneas se leerán desde la entrada estándar y queremos procesarlas una por una hasta que se agoten, una forma común en Python es utilizar un bucle infinito y capturar la excepción EOFError.

while True:
    try:
        x, y = map(int, input().split())
        print(abs(x - y))
```

```
    except EOFError:
        break

# Otro enfoque sería leer toda la entrada de una vez antes de comenzar a procesarla. Esto es adecuado cuando la entrada no es muy grande.

import sys

for line in sys.stdin.readlines():
     [x, y] = list(map(int, line.split()))
     print(abs(x - y))
```

2. Matemáticas

gcd.py

```
# Máximo común divisor (GCD)
import math
math.gcd(a, b, c)
```

lcm.py

```
# Mínimo común múltiplo (LCM)
import math
math.lcm(a, b, c)
```

sieve.py

```
# Criba de Erastótenes
# Algoritmo para generar números primos
# Sacado de: https://community.lambdatest.com/t/how-can-i-optimize-the-sieve-of-eratosthenes-in-python-for-larger-limits/34557/3

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]]
```

binary-exp.py

```
# Exponenciación rápida
# Calcula a^b mod q de manera eficiente O(log b).
pow(a, b, q)
```

binomial.py

```
# Coeficientes binomiales
math.comb(n, k)
```

3. Estructuras de Datos

Implementaciones de estructuras de datos no estándar que estén implementadas en la librería estándar

bst.py

```
# Árbol de Búsqueda Binaria (BST)
#Permiten buscar elementos en ellos en tiempo logarítmico. Esencialmente es como realizar una búsqueda binaria
# en una lista ordenada. Sus elementos deben tener un orden parcial.

class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

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

    def insert(self, key):
        def _insert(node, key):
            if not node:
                return Node(key)
            if key < node.key:
                node.left = _insert(node.left, key)
            elif key > node.key:
                node.right = _insert(node.right, key)
            return node

        self.root = _insert(self.root, key)

    def search(self, key):
        def _search(node, key):
            if not node or node.key == key:
                return node
            if key < node.key:
                return _search(node.left, key)
            else:
                return _search(node.right, key)

        return _search(self.root, key)

    def delete(self, key):
        def _min_value_node(node):
            current = node
            while current.left:
                current = current.left
            return current

        def _delete(node, key):
            if not node:
                return None

            if key < node.key:
                node.left = _delete(node.left, key)
            elif key > node.key:
                node.right = _delete(node.right, key)
            else:
                  # Caso 1: El nodo no tiene hijos o tiene un solo hijo.
                if not node.left:
                    return node.right
                elif not node.right:
                    return node.left

                # Caso 2: El nodo tiene dos hijos.
                 # Encontramos el sucesor inorden (el nodo más pequeño en el subárbol derecho).
                temp = _min_value_node(node.right)

                 # Copiamos el contenido del sucesor
```

```
inorden a este nodo.
                node.key = temp.key

                # Eliminamos el sucesor inorden.
            node.right = _delete(node.right, temp.key)

            return node

        self.root = _delete(self.root, key)
```

trie.py

```
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

    def print_trie(self, node=None, prefix=''):
      '''Solo sirve de ayuda para visualizar el trie'''
        if node is None:
            node = self.root

                            for    char,    child    in
sorted(node.children.items()):
                marker = '*' if child.is_end_of_word
else ''
              print('   ' * len(prefix) + f'- {char}
{marker}')
            self.print_trie(child, prefix + char)
```

segment_tree.py

```
#Es una estructura empleada para optimizar operaciones
sobre rangos (segmentos) de un array.
# Hay que modificar las cosas que son necesarias

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)
```

fenwick_tree.py

```
# Fenwick Tree

# Un árbol Fenwick funciona de manera similar a un
árbol segmentado,
```

```
# pero es menos potente, ya que la operación debe ser
#  inversible:  suma,  recuento  de  frecuencia...
funcionan, pero min/-
# max no. La única ventaja real es que es más rápido
#  de  escribir  y  ocupa  menos  espacio  (ambos  son
lineales).

class FenwickTree:
    def __init__(self, size):
        self.n = size
        self.tree = [0] * (self.n + 1)

    def update(self, i, delta):
        i += 1
        while i <= self.n:
            self.tree[i] += delta # this is modified
to change the operation
            i += i & -i

    def query(self, i):
        i += 1
        res = 0
        while i > 0:
            res += self.tree[i] #this is modified to
change the operation
            i -= i & -i
        return res

    def range_query(self, l, r):
         return self.query(r) - self.query(l - 1) #
This is modified to change the operation
```

4. Grafos

Un grafo G=(V,E) es un conjunto de vértices V y aristas (E, que almacena la información de conectividad entre los vértices en V).

Lectura de Grafos: Existen diferentes estructuras de datos para almacenar grafos, no obstante, la más empleada es la lista de Adyacencia, que abreviaremos como AL. En caso de ver la nomenclatura AM, nos estamos refiriendo a la matriz de adyacencia.

read_graphs.py

```
def leer_grafo_dirigido_ponderado(V, E):
    """Lee un grafo dirigido y ponderado."""
    AL = [[] for _ in range(V)]
    for _ in range(E):
      u, v, w = map(int, sys.stdin.readline().split())
        AL[u].append((v, w))  # Solo dirección u -> v
con peso w
    return AL

def leer_grafo_dirigido_no_ponderado(V, E):
    """Lee un grafo dirigido y no ponderado."""
    AL = [[] for _ in range(V)]
    for _ in range(E):
        u, v = map(int, sys.stdin.readline().split())
          AL[u].append(v)  # Solo dirección u -> v
sin peso
    return AL

def leer_grafo_no_dirigido_ponderado(V, E):
    """Lee un grafo no dirigido y ponderado."""
    AL = [[] for _ in range(V)]
    for _ in range(E):
      u, v, w = map(int, sys.stdin.readline().split())
        AL[u].append((v, w))  # u -> v con peso w
        AL[v].append((u, w))  # v -> u con peso w
    return AL

def leer_grafo_no_dirigido_no_ponderado(V, E):
    """Lee un grafo no dirigido y no ponderado."""
    AL = [[] for _ in range(V)]
    for _ in range(E):
        u, v = map(int, sys.stdin.readline().split())
        AL[u].append(v)  # u -> v sin peso
        AL[v].append(u)  # v -> u sin peso
    return AL
```

Dijkstra: Se utiliza para encontrar el camino más corto desde un nodo de inicio hasta todos los demás nodos en un grafo ponderado.

dijkstra.py

```
from heapq import heappush, heappop

def dijkstra(
    al: List[List[Tuple[int, int]]], s: int,
    s: int,
):
    """
    Ejecuta Dijkstra desde el nodo `s` en un grafo
    con lista de adyacencias `al`.

      // Sacado de https://github.com/stevenhalim/
cpbook-code/blob/master/ch4/sssp/dijkstra.py

    """

    dist = [float("inf")] * len(al)
    dist[s] = 0
    pq = [(0, s)]
    while 0 < len(pq):
        d, u = heappop(pq)

        if d > dist[u]:
            continue

        for v, w in al[u]:
            if dist[u] + w < dist[v]:
                dist[v] = dist[u] + w
                heappush(pq, (dist[v], v))

    return dist
```

DFS: Recorre todos los nodos de un grafo o árbol profundizando en cada rama antes de retroceder.

dfs.py

```
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)
```

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

bfs.py

```
from collections import deque
```

```python
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:  # un deque vacío se considera False
        node = to_visit.pop()
        for neighbor in graph[node]:
            if dist[neighbor] == float('inf'):
                dist[neighbor] = dist[node] + 1
                prec[neighbor] = node
                to_visit.append(neighbor)

    return dist, prec
```

Bellman-Ford: Calcula los caminos más cortos desde s en un grafo que puede tener aristas con pesos negativos.

- Los nodos inalcanzables obtienen `dist = inf`; los nodos alcanzables a través de ciclos de peso negativo obtienen `dist = -inf`.
- Se asume que $V^2 \cdot \max|w_i| < 2^{63}$.

bellman_ford.py

```python
# Bellman-Ford
def bellman_ford(graph, weight, source=0):
    n = len(graph)
    dist = [float('inf')] * n
    prec = [None] * n
    dist[source] = 0

    for _ in range(n):
        changed = False
        for node in range(n):
            for neighbor in graph[node]:
            alt = dist[node] + weight[node][neighbor]
                if alt < dist[neighbor]:
                    dist[neighbor] = alt
                    prec[neighbor] = node
                    changed = True
        if not changed:  # punto fijo alcanzado
             return dist, prec, False  # False -> no hay ciclo negativo

   return dist, prec, True  # True -> hay ciclo negativo
```

FloydWarshall: Calcula las distancias más cortas entre todos los pares de un grafo dirigido que podría tener aristas con pesos negativos.

- La entrada es una matriz de distancias m, donde m[i][j] = inf si i y j no son adyacentes.
- Como salida, m[i][j] se establece en la distancia más corta entre i y j, inf si no existe camino, o -inf si el camino pasa por un ciclo de peso negativo.

floyd_warshall.py

```python
import sys

input = sys.stdin.readline


def leer_grafo_floyd(n: int, e: int):
    am = [
        [float("inf") for _ in range(n)]
        for _ in range(n)
    ]
    for u in range(n):
        am[u][u] = 0
    for _ in range(e):
        u, v, w = map(int, input().split())
        # Se guarda la menor distancia si hay
        # aristas repetidas
        am[u][v] = min(am[u][v], w)
    return am


def floyd_warshall(am: list[list[int]], n: int):
    """Careful! This modifies am."""

    for k in range(n):
        for u in range(n):
            for v in range(n):
                am[u][v] = min(
                    am[u][v],
                    am[u][k] + am[k][v]
                )
    return am
```

EdmondsKarp: Algoritmo de flujo con complejidad garantizada $O(VE^2)$. Para obtener los valores de flujo de las aristas, compara las capacidades antes y después, y toma solo los valores positivos.

edmonds_karp.py

```python
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. Geometría

convex_hull.py

```
# Convex Hull (Envolvente Convexa)
#
# El objetivo de este algoritmo es encontrar los puntos de un conjunto
# que forman el perímetro de la figura convexa más pequeña que contiene
# a todos los demás puntos. Piensa en ello como si estuvieras estirando
# una banda elástica alrededor de un conjunto de clavos. Los puntos
# tocados por la banda elástica son la envolvente convexa.

def convex_hull(points):
    if len(points) <= 1:
        return points

    # Sort by x, then y
    points = sorted(list(set(points)))

    def cross(o, a, b):
        # Cross product of vectors OA and OB
        return (a[0] - o[0]) * (b[1] - o[1]) - (a[1] - o[1]) * (b[0] - o[0])

    lower = []
    for p in points:
          while len(lower) >= 2 and cross(lower[-2], lower[-1], p) < 0: # Add <= here to make it non strict
              lower.pop()
        lower.append(p)
    upper = []
    for p in reversed(points):
          while len(upper) >= 2 and cross(upper[-2], upper[-1], p) < 0:  # Add <= here to make it non strict
              upper.pop()
        upper.append(p)

    # Concatenate lower and upper, removing the last point of each (duplicate)
    return lower[:-1] + upper[:-1]
```

6. Apéndices

techniques.txt

```
Recursión
  Dividir y conquistar
    Encontrar puntos interesantes en $N \log N$
Análisis de algoritmos
  Teorema maestro
  Complejidad temporal amortizada
Algoritmo voraz (Greedy)
  Planificación
  Suma máxima de subvector contiguo
  Invariantes
  Codificación Huffman
Teoría de grafos
  Grafos dinámicos (mantenimiento extra)
  Búsqueda en anchura (BFS)
  Búsqueda en profundidad (DFS)
    * Árboles normales / Árboles DFS
  Algoritmo de Dijkstra
  AGM: Algoritmo de Prim
  Bellman-Ford
  Teorema de König y cobertura de vértices
  Flujo máximo de costo mínimo
  Conmutación de Lovász
  Teorema de matrices de árboles
  Emparejamiento máximo, grafos generales
  Hopcroft-Karp
  Teorema de matrimonio de Hall
  Secuencias gráficas
  Floyd-Warshall
  Ciclos de Euler
  Redes de flujo
    * Caminos aumentantes
    * Edmonds-Karp
  Emparejamiento bipartito
  Cobertura mínima de caminos
  Ordenamiento topológico
  Componentes fuertemente conectados
  2-SAT
   Vértices de corte, aristas de corte y componentes biconectadas
  Coloración de aristas
    * Árboles
  Coloración de vértices
    * Grafos bipartitos (=> árboles)
    * $3^n$ (caso especial de set cover)
  Diámetro y centroide
  $k$-ésimo camino más corto
  Ciclo más corto
Programación dinámica
  Mochila
  Cambio de monedas
  Subsecuencia común más larga
  Subsecuencia creciente más larga
  Número de caminos en un DAG
  Camino más corto en un DAG
  Programación dinámica sobre intervalos
  Programación dinámica sobre subconjuntos
  Programación dinámica sobre probabilidades
  Programación dinámica sobre árboles
  $3^n$ set cover
  Dividir y conquistar
  Optimización de Knuth
  Optimización de convex hull
  RMQ (Sparse Table o saltos $2^k$)
  Ciclo bitónico
   Particionamiento logarítmico (loop sobre el más restringido)
Combinatoria
  Cálculo de coeficientes binomiales
  Principio del palomar
  Inclusión/exclusión
  Números de Catalan
  Teorema de Pick
Teoría de números
  Partes enteras
  Divisibilidad
  Algoritmo euclidiano
  Aritmética modular
    * Multiplicación modular
    * Inversos modulares
    * Exponenciación modular por cuadrados
  Teorema chino del resto
  Pequeño teorema de Fermat
  Teorema de Euler
```

- Función Phi
- Número de Frobenius
- Reciprocidad cuadrática
- Pollard-Rho
- Miller-Rabin
- Levantamiento de Hensel
- Saltos de raíces de Vieta

Teoría de juegos
- Juegos combinatorios
- Árboles de juego
- Minimax
- Nim
- Juegos sobre grafos
- Juegos sobre grafos con ciclos
- Números de Grundy
- Juegos bipartitos sin repetición
- Juegos generales sin repetición
- Poda alfa-beta

Teoría de la probabilidad

Optimización
- Búsqueda binaria
- Búsqueda ternaria
- Unimodalidad y funciones convexas
- Búsqueda binaria sobre derivadas

Métodos numéricos
- Integración numérica
- Método de Newton
- Búsqueda de raíces con búsqueda binaria/ternaria
- Búsqueda de la sección áurea

Matrices
- Eliminación gaussiana
- Exponenciación por cuadrados

Ordenamiento
- Radix sort

Geometría
- Coordenadas y vectores
  * Producto cruz
  * Producto escalar
- Convex hull
- Corte de polígonos
- Par más cercano
- Compresión de coordenadas
- Quadtrees
- KD-trees
- Todas las intersecciones segmento-segmento

Barrido (Sweeping)
- Discretización (convertir en eventos y barrer)
- Barrido por ángulos
- Barrido de líneas
- Segundas derivadas discretas

Cadenas (Strings)
- Subcadena común más larga
- Subsecuencias palíndromas
- Knuth-Morris-Pratt
- Tries
- Hashes polinomiales rodantes
- Array de sufijos
- Árbol de sufijos
- Aho-Corasick
- Algoritmo de Manacher
- Listas de posiciones de letras

Búsqueda combinatoria
- Meet in the middle
- Fuerza bruta con poda
- Mejor primero (A*)
- Búsqueda bidireccional
- DFS/A* con profundización iterativa

Estructuras de datos
- LCA (saltos 2^k en árboles en general)
- Técnica pull/push en árboles
- Descomposición heavy-light
- Descomposición por centroides
- Propagación perezosa (Lazy propagation)
- Árboles auto-balanceados
- Convex hull trick (wcipeg.com/wiki/Convex_hull_trick)
- Colas/Stacks monotónicos / colas deslizantes
- Cola deslizante usando 2 stacks
- Segment tree persistente