==

💻 EJERCICIOS Y PROYECTOS PRÁCTICOS - MATEMÁTICAS PARA PROGRAMACIÓN

==

Por: Ernesto

Objetivo: Práctica intensiva para dominar cada concepto

Estructura: Ejercicios progresivos + Proyectos integradores

==

==

📚 ÍNDICE DE CONTENIDOS

==

PARTE 1: MATEMÁTICA DISCRETA
├─ 1.1 Árboles - Ejercicios y Proyectos
├─ 1.2 Grafos - Ejercicios y Proyectos
├─ 1.3 Combinatoria - Ejercicios y Proyectos
├─ 1.4 Conjuntos - Ejercicios y Proyectos
├─ 1.5 Lógica - Ejercicios y Proyectos
└─ 1.6 Proyecto Integrador Discreta

PARTE 2: ÁLGEBRA LINEAL
├─ 2.1 Vectores - Ejercicios y Proyectos
├─ 2.2 Matrices - Ejercicios y Proyectos
├─ 2.3 Transformaciones - Ejercicios y Proyectos
├─ 2.4 Eigenvalues - Ejercicios y Proyectos
└─ 2.5 Proyecto Integrador Álgebra

PARTE 3: PROBABILIDAD Y ESTADÍSTICA
├─ 3.1 Probabilidad - Ejercicios y Proyectos
├─ 3.2 Distribuciones - Ejercicios y Proyectos
├─ 3.3 Estadística - Ejercicios y Proyectos
├─ 3.4 Inferencia - Ejercicios y Proyectos
└─ 3.5 Proyecto Integrador Probabilidad

PARTE 4: CÁLCULO Y OPTIMIZACIÓN
├─ 4.1 Derivadas - Ejercicios y Proyectos
├─ 4.2 Integrales - Ejercicios y Proyectos
├─ 4.3 Optimización - Ejercicios y Proyectos
└─ 4.4 Proyecto Integrador Cálculo

PARTE 5: PROYECTOS FINALES INTEGRADORES

==

PARTE 1: MATEMÁTICA DISCRETA

==

1.1 ÁRBOLES - EJERCICIOS PROGRESIVOS

NIVEL 1: FUNDAMENTOS (20 ejercicios)

EJERCICIOS TEÓRICOS (papel y lápiz):

1. Dado el siguiente árbol, encontrar:
   a) Preorden
   b) Inorden
   c) Postorden

```
      A
     / \
    B   C
   / \   \
  D   E   F
```

2. Construir el árbol dados los recorridos:
   Preorden: F, B, A, D, C, E, G, I, H
   Inorden: A, B, C, D, E, F, G, H, I

3. ¿Es válido un árbol con estos recorridos?
   Preorden: A, B, D, E, C, F
   Inorden: D, B, E, A, C, F
   Postorden: D, E, B, F, A, C
   (Verificar consistencia)

4-10. [Ejercicios similares con diferentes árboles]
   - Con operadores matemáticos (+, -, *, /)
   - Con funciones (sen, cos, ^, !)
   - Árboles desbalanceados
   - Árboles completos
   - Árboles degenerados (lista)

11. Altura de árboles:
    Calcular la altura de cada árbol del ejercicio anterior.

12-15. Árboles Binarios de Búsqueda (BST):
    Insertar la siguiente secuencia en un BST vacío:
    [50, 30, 70, 20, 40, 60, 80]

    a) Dibujar el árbol resultante
    b) Mostrar recorridos
    c) Buscar el valor 40 (paso a paso)
    d) Eliminar el valor 30 (mostrar casos)

16-20. Problemas de conteo:

- ¿Cuántos BSTs diferentes se pueden formar con {1,2,3}?
- ¿Cuántos árboles binarios completos de altura 3?
- Nodos en árbol completo de altura h
- Hojas en árbol binario completo
- Nodos internos vs nodos hoja

EJERCICIOS DE PROGRAMACIÓN:

EJERCICIO P1: Implementar Árbol Binario

```python
# Archivo: binary_tree.py

class Node:
    """
    TODO: Implementar clase Node
    - __init__(self, data)
    - Atributos: data, left, right
    """
    pass

class BinaryTree:
    """
    TODO: Implementar BinaryTree con los siguientes métodos:

    1. __init__(self)
    2. insert_left(self, current, data)
    3. insert_right(self, current, data)
    4. preorder(self, node) -> list
    5. inorder(self, node) -> list
    6. postorder(self, node) -> list
    7. height(self, node) -> int
    8. count_nodes(self, node) -> int
    9. count_leaves(self, node) -> int
    10. is_balanced(self, node) -> bool
    """
    pass

# Tests
def test_binary_tree():
    tree = BinaryTree()
    # TODO: Crear el árbol del ejercicio 1
    # TODO: Verificar recorridos
    # TODO: Verificar altura
    # TODO: Contar nodos y hojas
    pass

if __name__ == "__main__":
    test_binary_tree()
```

EJERCICIO P2: Implementar BST

```python
# Archivo: bst.py

class BST:
    """
    TODO: Implementar Binary Search Tree

    Métodos requeridos:
    1. insert(self, data)
    2. search(self, data) -> bool
    3. find_min(self) -> data
    4. find_max(self) -> data
    5. delete(self, data)
    6. inorder_successor(self, node)
    7. is_valid_bst(self) -> bool
    """
    pass

# Tests específicos
def test_bst_operations():
    bst = BST()

    # Test 1: Inserción
    values = [50, 30, 70, 20, 40, 60, 80]
    for val in values:
        bst.insert(val)

    # TODO: Verificar estructura correcta
    # TODO: Buscar valores existentes y no existentes
    # TODO: Encontrar min y max
    # TODO: Eliminar nodos (casos: hoja, 1 hijo, 2 hijos)
    # TODO: Verificar que sigue siendo BST válido
    pass
```

EJERCICIO P3: Reconstrucción de Árboles

```python
# Archivo: tree_reconstruction.py

def build_tree_pre_in(preorder, inorder):
    """
    Reconstruir árbol binario dados preorden e inorden.

    Args:
        preorder: List[int] - recorrido en preorden
        inorder: List[int] - recorrido en inorden

    Returns:
        Node - raíz del árbol reconstruido
```

```
    Ejemplo:
        preorder = [3,9,20,15,7]
        inorder = [9,3,15,20,7]

        Debe retornar:
            3
           / \
          9  20
            /  \
           15   7
    """
    # TODO: Implementar
    pass

def build_tree_post_in(postorder, inorder):
    """
    Reconstruir árbol binario dados postorden e inorden.
    """
    # TODO: Implementar
    pass

# Tests
def test_reconstruction():
    # Caso 1: Árbol simple
    pre = [1, 2, 4, 5, 3]
    ino = [4, 2, 5, 1, 3]
    tree = build_tree_pre_in(pre, ino)
    # TODO: Verificar que los recorridos del árbol construido
    #       coinciden con los originales

    # Caso 2: Árbol con expresión matemática
    pre = ['+', '*', 'a', 'b', 'c']
    ino = ['a', '*', 'b', '+', 'c']
    # TODO: Construir y evaluar

    pass
```

EJERCICIO P4: Visualizador de Árboles

```python
# Archivo: tree_visualizer.py

import matplotlib.pyplot as plt
import matplotlib.patches as patches

class TreeVisualizer:
    """
    Visualizador gráfico de árboles binarios.
```

```
    TODO: Implementar métodos:
    1. draw_tree(root)
    2. calculate_positions(node, x, y, layer)
    3. draw_node(x, y, data)
    4. draw_edge(x1, y1, x2, y2)
    5. animate_traversal(root, traversal_type)
       - Mostrar paso a paso el recorrido
       - Resaltar nodo actual
    """

    def draw_tree(self, root):
        """
        Dibuja el árbol completo.

        TODO:
        - Calcular posiciones de nodos
        - Dibujar nodos como círculos
        - Dibujar aristas como líneas
        - Etiquetar nodos con sus valores
        """
        pass

    def animate_preorder(self, root):
        """
        Anima el recorrido en preorden.

        TODO:
        - Crear frames de animación
        - Resaltar nodo visitado
        - Mostrar orden de visita
        - Guardar como GIF
        """
        pass

# Ejemplo de uso
if __name__ == "__main__":
    # TODO: Crear árbol de ejemplo
    # TODO: Visualizar
    # TODO: Animar recorridos
    pass
```

EJERCICIO P5: Árbol de Expresiones

```python
# Archivo: expression_tree.py

class ExpressionTree:
    """
    Árbol para representar expresiones matemáticas.
```

```
TODO: Implementar
1. build_from_postfix(expression)
2. build_from_prefix(expression)
3. build_from_infix(expression)
4. evaluate() -> float
5. to_infix() -> str (con paréntesis mínimos)
6. to_postfix() -> str
7. derivative(variable) -> ExpressionTree
   (Bonus: calcular derivada simbólica)
"""

def build_from_postfix(self, postfix):
    """
    Construir árbol desde notación postfija.

    Ejemplo:
        Input: "3 4 + 2 * 7 /"
        Árbol:
              /
             / \
            *   7
           / \
          +   2
         / \
        3   4
    """
    # TODO: Usar pila para construir
    pass

def evaluate(self):
    """
    Evaluar la expresión representada por el árbol.

    TODO:
    - Caso base: nodo hoja (número)
    - Caso recursivo: evaluar hijos y aplicar operador
    """
    pass

def derivative(self, var='x'):
    """
    BONUS AVANZADO: Calcular derivada simbólica.

    Reglas:
    - d/dx(c) = 0
    - d/dx(x) = 1
    - d/dx(u + v) = du/dx + dv/dx
    - d/dx(u * v) = u*dv/dx + v*du/dx
    - etc.
    """
```

```
        pass

# Tests
def test_expression_tree():
    # Test 1: Construcción y evaluación
    expr = ExpressionTree()
    expr.build_from_postfix("3 4 + 2 * 7 /")
    result = expr.evaluate()
    assert result == 2.0, f"Esperado 2.0, obtenido {result}"

    # Test 2: Conversión entre notaciones
    infix = expr.to_infix()
    assert infix == "((3 + 4) * 2) / 7"

    # Test 3: Expresión con funciones
    expr2 = ExpressionTree()
    expr2.build_from_prefix("+ sen x cos x")
    # Debe crear: sen(x) + cos(x)

    # TODO: Más tests
    pass
```

NIVEL 2: INTERMEDIO (15 ejercicios)

EJERCICIO P6: AVL Tree (Balanceo Automático)

```python
# Archivo: avl_tree.py

class AVLNode:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
        self.height = 1  # Altura del nodo

class AVLTree:
    """
    Árbol AVL - BST auto-balanceado.

    TODO: Implementar
    1. get_height(node) -> int
    2. get_balance(node) -> int
    3. rotate_right(y)
    4. rotate_left(x)
    5. insert(data)
       - Insertar como BST
       - Actualizar alturas
```

```
    - Balancear si es necesario
  6. delete(data)
  7. visualize_rotations()
    - Mostrar gráficamente las rotaciones
  """

  def rotate_right(self, y):
    """
    Rotación a la derecha:

       y                x
      / \              / \
     x   C    -->     A   y
    / \                  / \
    A   B                B   C
    """
    # TODO: Implementar
    pass

  def rotate_left(self, x):
    """
    Rotación a la izquierda (espejo de right)
    """
    # TODO: Implementar
    pass

  def insert(self, data):
    """
    Insertar y balancear.

    Casos de balanceo:
    1. Left-Left: rotate_right(node)
    2. Right-Right: rotate_left(node)
    3. Left-Right: rotate_left(node.left), rotate_right(node)
    4. Right-Left: rotate_right(node.right), rotate_left(node)
    """
    # TODO: Implementar
    pass

# Tests
def test_avl():
  avl = AVLTree()

  # Caso que causa desbalanceo Left-Left
  for val in [3, 2, 1]:
    avl.insert(val)
  # Verificar que se rotó correctamente

  # Caso que causa desbalanceo Right-Right
  avl2 = AVLTree()
```

```
    for val in [1, 2, 3]:
        avl2.insert(val)

    # TODO: Más tests con diferentes patrones
    # TODO: Verificar que la altura siempre es O(log n)
    pass
```

EJERCICIO P7: Heap (Min-Heap y Max-Heap)

```python
# Archivo: heap.py

class MinHeap:
    """
    Min-Heap implementado como array.

    Propiedades:
    - Árbol binario completo
    - Cada padre es menor que sus hijos
    - Representación: array[i] tiene hijos en 2i+1 y 2i+2

    TODO: Implementar
    1. insert(data)
    2. extract_min() -> data
    3. peek_min() -> data
    4. heapify_up(index)
    5. heapify_down(index)
    6. build_heap(array)
    7. heap_sort(array) -> sorted_array
    8. visualize() - mostrar como árbol
    """

    def __init__(self):
        self.heap = []

    def parent(self, i):
        return (i - 1) // 2

    def left_child(self, i):
        return 2 * i + 1

    def right_child(self, i):
        return 2 * i + 2

    def insert(self, data):
        """
        1. Agregar al final
        2. Heapify up
        """
        # TODO: Implementar
```

```
        pass

    def extract_min(self):
        """
        1. Guardar raíz
        2. Mover último elemento a raíz
        3. Heapify down
        4. Retornar raíz guardada
        """
        # TODO: Implementar
        pass

    def heapify_up(self, i):
        """
        Subir elemento hasta que se cumpla propiedad de heap.
        """
        # TODO: Implementar
        pass

    def heapify_down(self, i):
        """
        Bajar elemento hasta que se cumpla propiedad de heap.
        """
        # TODO: Implementar
        pass

class MaxHeap(MinHeap):
    """
    Max-Heap: invertir comparaciones del Min-Heap.
    """
    # TODO: Heredar y modificar comparaciones
    pass

# Tests
def test_heap():
    heap = MinHeap()

    # Test inserción
    values = [5, 3, 7, 1, 9, 2]
    for val in values:
        heap.insert(val)

    # Test extracción (debe salir ordenado)
    result = []
    while heap.heap:
        result.append(heap.extract_min())

    assert result == sorted(values)

    # TODO: Test heap sort
```

```
    # TODO: Test con duplicados
    # TODO: Test casos borde
    pass
```

EJERCICIO P8: Trie (Prefix Tree)

```python
# Archivo: trie.py

class TrieNode:
    def __init__(self):
        self.children = {}  # dict: char -> TrieNode
        self.is_end_of_word = False
        self.frequency = 0  # para autocomplete

class Trie:
    """
    Trie para búsqueda eficiente de strings.

    Aplicaciones:
    - Autocomplete
    - Spell checker
    - IP routing
    - Dictionary

    TODO: Implementar
    1. insert(word)
    2. search(word) -> bool
    3. starts_with(prefix) -> bool
    4. delete(word)
    5. autocomplete(prefix) -> List[str]
    6. count_words_with_prefix(prefix) -> int
    7. longest_common_prefix() -> str
    8. visualize()
    """

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

    def insert(self, word):
        """
        Insertar palabra en el Trie.

        Ejemplo: insert("cat")
        root -> c -> a -> t (is_end=True)
        """
        # TODO: Implementar
        pass

    def search(self, word):
```

```
        """
        Buscar palabra completa.

        Retorna True solo si:
        1. El path existe
        2. is_end_of_word es True
        """
        # TODO: Implementar
        pass

    def autocomplete(self, prefix):
        """
        Retornar todas las palabras que empiezan con prefix.

        Algoritmo:
        1. Navegar hasta el nodo del prefix
        2. DFS desde ese nodo para colectar palabras
        3. Ordenar por frecuencia (opcional)
        """
        # TODO: Implementar
        pass

# Tests
def test_trie():
    trie = Trie()

    # Diccionario de prueba
    words = ["cat", "cats", "dog", "dogs", "car", "card", "care"]
    for word in words:
        trie.insert(word)

    # Test búsqueda
    assert trie.search("cat") == True
    assert trie.search("ca") == False
    assert trie.starts_with("ca") == True

    # Test autocomplete
    suggestions = trie.autocomplete("ca")
    assert set(suggestions) == {"cat", "cats", "car", "card", "care"}

    # TODO: Test delete
    # TODO: Test con strings largos
    # TODO: Comparar performance vs list
    pass

# PROYECTO BONUS: Spell Checker
def build_spell_checker():
    """
    Usar Trie + edit distance para spell checking.
```

```
    1. Cargar diccionario en Trie
    2. Para palabra incorrecta:
       - Buscar palabras a distancia 1 o 2 (Levenshtein)
       - Sugerir las más probables
    """
    # TODO: Implementar
    pass
```

EJERCICIO P9: Segment Tree

```python
# Archivo: segment_tree.py

class SegmentTree:
    """
    Árbol de segmentos para queries de rango eficientes.

    Operaciones en O(log n):
    - Query: suma/min/max en rango [L, R]
    - Update: modificar un elemento

    Aplicaciones:
    - Range sum queries
    - Range minimum/maximum
    - Lazy propagation para updates de rango

    TODO: Implementar
    1. build(array)
    2. query(L, R) -> resultado en rango
    3. update(index, value)
    4. range_update(L, R, value) - BONUS
    """

    def __init__(self, array, operation='sum'):
        """
        operation: 'sum', 'min', 'max', 'gcd'
        """
        self.n = len(array)
        self.operation = operation
        self.tree = [0] * (4 * self.n)
        self.build(array, 0, 0, self.n - 1)

    def build(self, array, node, start, end):
        """
        Construir árbol recursivamente.

        Árbol de segmentos para [1, 3, 2, 7, 9, 11]:
                    33
                  /    \
                13      20
```

```
            / \   / \
           4   9 16  11
          / \ / \
         1  3 2  7
        """
        # TODO: Implementar
        pass

    def query(self, L, R):
        """
        Consultar operación en rango [L, R].
        """
        # TODO: Implementar query recursivo
        pass

    def update(self, index, value):
        """
        Actualizar elemento en index.
        """
        # TODO: Implementar update recursivo
        pass

# Tests
def test_segment_tree():
    array = [1, 3, 5, 7, 9, 11]

    # Test sum queries
    st_sum = SegmentTree(array, 'sum')
    assert st_sum.query(1, 3) == 15  # 3 + 5 + 7

    # Test update
    st_sum.update(1, 10)
    assert st_sum.query(1, 3) == 22  # 10 + 5 + 7

    # Test min queries
    st_min = SegmentTree(array, 'min')
    assert st_min.query(1, 4) == 3

    # TODO: Test max
    # TODO: Test casos borde
    # TODO: Benchmark vs solución naive
    pass
```

EJERCICIO P10: Fenwick Tree (Binary Indexed Tree)

```python
# Archivo: fenwick_tree.py

class FenwickTree:
    """
```

```
    BIT para prefix sums eficientes.

    Operaciones en O(log n):
    - update(index, delta): incrementar elemento
    - prefix_sum(index): suma de [0, index]
    - range_sum(L, R): suma de [L, R]

    Ventaja vs Segment Tree: más simple, menos memoria.

    TODO: Implementar
    1. update(index, delta)
    2. prefix_sum(index)
    3. range_sum(L, R)
    4. point_query(index)
    """

    def __init__(self, n):
        self.n = n
        self.tree = [0] * (n + 1)  # 1-indexed

    def update(self, index, delta):
        """
        Incrementar array[index] en delta.

        Propaga hacia arriba usando:
        index += index & (-index)
        """
        # TODO: Implementar
        pass

    def prefix_sum(self, index):
        """
        Retorna suma de array[0:index+1].

        Acumula bajando usando:
        index -= index & (-index)
        """
        # TODO: Implementar
        pass

    def range_sum(self, left, right):
        """
        Suma en rango [left, right].

        Usa: prefix_sum(right) - prefix_sum(left-1)
        """
        return self.prefix_sum(right) - self.prefix_sum(left - 1)

# Aplicación: Counting Inversions
def count_inversions(array):
```

```
    """
    Contar inversiones en array usando Fenwick Tree.

    Inversión: par (i, j) donde i < j pero array[i] > array[j]

    Ejemplo: [5, 2, 6, 1] tiene inversiones:
    - (5, 2), (5, 1), (2, 1), (6, 1) = 4 inversiones
    """
    # TODO: Implementar usando Fenwick Tree
    # Hint: coordinar compresión + Fenwick
    pass
```

NIVEL 3: AVANZADO (10 ejercicios)

EJERCICIO P11: Red-Black Tree

```python
# Archivo: red_black_tree.py

class Color:
    RED = 0
    BLACK = 1

class RBNode:
    def __init__(self, data, color=Color.RED):
        self.data = data
        self.color = color
        self.left = None
        self.right = None
        self.parent = None

class RedBlackTree:
    """
    Red-Black Tree - BST balanceado con garantías estrictas.

    Propiedades:
    1. Cada nodo es RED o BLACK
    2. Raíz es BLACK
    3. Hojas (NIL) son BLACK
    4. Si nodo es RED, sus hijos son BLACK
    5. Todos los caminos de nodo a hoja tienen mismo # de nodos BLACK

    Garantiza: altura ≤ 2 log(n+1)

    TODO: Implementar (DESAFÍO AVANZADO)
    1. insert(data)
    2. insert_fixup(node)
    3. delete(data)
```

```
    4. delete_fixup(node)
    5. rotate_left(node)
    6. rotate_right(node)
    7. verify_properties() -> bool
    """

    # Este es el árbol balanceado más complejo
    # Tomará tiempo dominarlo
    # TODO: Implementar paso a paso
    pass
```

EJERCICIO P12: B-Tree

```python
# Archivo: b_tree.py

class BTreeNode:
    def __init__(self, t, leaf=True):
        """
        t: grado mínimo
        node puede tener entre t-1 y 2t-1 keys
        """
        self.t = t
        self.keys = []
        self.children = []
        self.leaf = leaf

class BTree:
    """
    B-Tree: árbol balanceado usado en bases de datos y filesystems.

    Propiedades:
    - Todos los nodos hoja están al mismo nivel
    - Node tiene entre t-1 y 2t-1 keys (excepto raíz)
    - Optimizado para I/O en disco

    TODO: Implementar
    1. search(key)
    2. insert(key)
    3. split_child(parent, i)
    4. delete(key)
    5. merge_children(parent, i)

    APLICACIÓN: Simular índice de base de datos
    """
    pass
```

EJERCICIO P13: Splay Tree

```python
```

```python
# Archivo: splay_tree.py

class SplayTree:
    """
    Splay Tree: BST auto-ajustable.

    Propiedad:
    - Elementos accedidos recientemente están cerca de la raíz
    - Operaciones amortizan a O(log n)
    - No requiere información adicional (sin colores, alturas)

    Operación clave: splay(node)
    - Rotar node hasta que sea la raíz
    - Casos: zig, zig-zig, zig-zag

    TODO: Implementar
    1. splay(node)
    2. insert(data)
    3. search(data)
    4. delete(data)

    EXPERIMENTO: Comparar con AVL en diferentes workloads
    """
    pass
```

EJERCICIO P14: Suffix Tree

```python
# Archivo: suffix_tree.py

class SuffixTree:
    """
    Suffix Tree: para pattern matching ultra-rápido.

    Construcción: O(n) con Ukkonen's algorithm
    Búsqueda de patrón: O(m) donde m = longitud del patrón

    Aplicaciones:
    - Búsqueda de substring
    - Longest common substring
    - Longest repeated substring
    - Compresión de datos

    TODO: Implementar
    1. build(text) - usar Ukkonen's algorithm
    2. search_pattern(pattern) -> List[int]
    3. longest_repeated_substring() -> str
    4. longest_common_substring(text1, text2) -> str

    NOTA: Implementación compleja, tomar tiempo
```

```
    """
    pass
```

PROYECTO INTEGRADOR: ÁRBOLES

PROYECTO: "TreeLab - Laboratorio Interactivo de Estructuras de Árbol"

DESCRIPCIÓN:
Plataforma web interactiva para experimentar con todas las estructuras de
árbol implementadas.

COMPONENTES:

1. VISUALIZADOR UNIVERSAL
   ├─ Dibujar cualquier tipo de árbol
   ├─ Animaciones de operaciones (insert, delete, rotate)
   ├─ Código de colores para diferentes tipos
   └─ Zoom, pan, reset

2. COMPARADOR DE ESTRUCTURAS
   ├─ Insertar misma secuencia en diferentes árboles
   ├─ Visualizar diferencias
   ├─ Comparar alturas
   └─ Benchmark de operaciones

3. MODE EDUCATIVO
   ├─ Tutorial paso a paso de cada estructura
   ├─ Visualización de rotaciones
   ├─ Quiz interactivo
   └─ Ejercicios guiados

4. SIMULADOR DE APLICACIONES
   ├─ Autocomplete con Trie
   ├─ Expresión evaluator con árbol de expresiones
   ├─ Range queries con Segment Tree
   └─ Database index con B-Tree

5. BENCHMARKING SUITE
   ├─ Generar datasets de prueba
   ├─ Medir tiempo de operaciones
   ├─ Graficar resultados
   ├─ Comparar Big-O teórico vs práctico
   └─ Reportes automatizados

TECNOLOGÍAS:
├─ Backend: Python (Flask)
├─ Frontend: HTML/CSS/JavaScript

├─ Visualización: D3.js o vis.js
├─ Charts: Chart.js o Plotly
└─ Deploy: Heroku o similar

FEATURES AVANZADAS (BONUS):
├─ Export de árboles a imágenes/SVG
├─ Share de árboles (generar URL)
├─ Competencias: resolver desafíos con árboles
└─ Integración con LeetCode (importar problemas)

ENTREGABLES:
☐ Código fuente completo en GitHub
☐ Web app deployada
☐ Documentación completa
☐ Video demo (5-10 min)
☐ Blog post explicando implementación

TIEMPO ESTIMADO: 3-4 semanas

1.2 GRAFOS - EJERCICIOS PROGRESIVOS

NIVEL 1: FUNDAMENTOS (30 ejercicios)

EJERCICIOS TEÓRICOS:

1-5. Representación de grafos:
Dado el siguiente grafo, representarlo como:
a) Matriz de adyacencia
b) Lista de adyacencia
c) Lista de aristas

Grafo ejemplo:
0 -- 1 -- 2
| | |
3 -- 4 -- 5

6-10. Propiedades básicas:
Para cada grafo, calcular:
- Número de vértices y aristas
- Grado de cada vértice
- Grafo es conexo?
- Grafo es bipartito?
- Contiene ciclos?

11-15. BFS manual:
Ejecutar BFS empezando desde vértice dado.

Mostrar:
- Orden de visita
- Distancias desde origen
- Árbol BFS
- Niveles

16-20. DFS manual:
Ejecutar DFS empezando desde vértice dado.
Mostrar:
- Orden de visita (preorden y postorden)
- Tiempos de discovery y finish
- Árbol DFS
- Back edges, forward edges, cross edges

21-25. Componentes conexas:
Encontrar todas las componentes conexas.

26-30. Grafos dirigidos:
- Calcular grado de entrada y salida
- Encontrar sumideros y fuentes
- Detectar ciclos
- Ordenamiento topológico (si es DAG)

EJERCICIOS DE PROGRAMACIÓN:

EJERCICIO G1: Implementar Grafo

```python
# Archivo: graph.py

from collections import deque, defaultdict
from typing import List, Set, Dict

class Graph:
    """
    Grafo usando lista de adyacencia.
    Soporta grafos dirigidos y no dirigidos, ponderados y no ponderados.

    TODO: Implementar todos los métodos
    """

    def __init__(self, directed=False, weighted=False):
        self.adj_list = defaultdict(list)
        self.directed = directed
        self.weighted = weighted
        self.num_vertices = 0
        self.num_edges = 0

    def add_vertex(self, v):
        """Agregar vértice."""
        # TODO: Implementar
```

```
        pass

    def add_edge(self, u, v, weight=1):
        """
        Agregar arista de u a v.
        Si no dirigido, agregar también v a u.
        """
        # TODO: Implementar
        pass

    def remove_edge(self, u, v):
        """Eliminar arista."""
        # TODO: Implementar
        pass

    def get_neighbors(self, v):
        """Retornar vecinos de v."""
        # TODO: Implementar
        pass

    def degree(self, v):
        """Grado del vértice v."""
        # TODO: Implementar
        # Para dirigido: separar in_degree y out_degree
        pass

    def vertices(self):
        """Retornar lista de vértices."""
        # TODO: Implementar
        pass

    def edges(self):
        """Retornar lista de aristas."""
        # TODO: Implementar
        pass

    def __str__(self):
        """Representación en string."""
        # TODO: Implementar formato legible
        pass

    def to_adjacency_matrix(self):
        """Convertir a matriz de adyacencia."""
        # TODO: Implementar
        # Retornar numpy array o lista de listas
        pass

# Tests básicos
def test_graph():
    # Test 1: Grafo no dirigido simple
```

```
    g = Graph(directed=False, weighted=False)

    edges = [(0, 1), (0, 3), (1, 2), (1, 4), (2, 5), (3, 4), (4, 5)]
    for u, v in edges:
        g.add_edge(u, v)

    assert g.num_edges == 7
    assert g.degree(1) == 3

    # Test 2: Grafo dirigido ponderado
    dg = Graph(directed=True, weighted=True)
    dg.add_edge(0, 1, 4)
    dg.add_edge(1, 2, 3)
    dg.add_edge(2, 0, 1)

    assert dg.in_degree(0) == 1
    assert dg.out_degree(0) == 1

    # TODO: Más tests
    pass
```

EJERCICIO G2: BFS y DFS

```python
# Archivo: graph_traversal.py

from collections import deque

def bfs(graph, start):
    """
    Breadth-First Search.

    Retorna:
        visited: conjunto de nodos visitados
        distances: diccionario {nodo: distancia desde start}
        parent: diccionario {nodo: padre en árbol BFS}

    Aplicaciones:
    - Camino más corto (sin pesos)
    - Componentes conexas
    - Bipartito test
    """
    visited = set()
    distances = {start: 0}
    parent = {start: None}
    queue = deque([start])

    while queue:
        # TODO: Implementar BFS
        pass
```

```
    return visited, distances, parent

def shortest_path_bfs(graph, start, goal):
    """
    Encontrar camino más corto de start a goal (sin pesos).
    Retornar el camino como lista de nodos.
    """
    visited, distances, parent = bfs(graph, start)

    if goal not in visited:
        return None  # No hay camino

    # Reconstruir camino desde goal hasta start
    path = []
    current = goal
    # TODO: Implementar reconstrucción

    return path[::-1]  # Invertir

def bfs_levels(graph, start):
    """
    BFS que retorna nodos agrupados por nivel.

    Retorna: [[nivel 0], [nivel 1], [nivel 2], ...]
    """
    # TODO: Implementar
    pass

def dfs_recursive(graph, node, visited=None):
    """
    DFS recursivo.

    Retorna: orden de visita (lista)
    """
    if visited is None:
        visited = set()

    # TODO: Implementar DFS recursivo
    pass

def dfs_iterative(graph, start):
    """
    DFS iterativo usando stack.
    """
    visited = set()
    stack = [start]
    order = []

    while stack:
```

```
        # TODO: Implementar DFS iterativo
        pass

    return order

def dfs_with_timestamps(graph, start):
    """
    DFS con tiempos de discovery y finish.

    Retorna:
        discovery: {nodo: tiempo de discovery}
        finish: {nodo: tiempo de finish}
        parent: {nodo: padre}
    """
    time = [0]  # Usar lista para poder modificar en función anidada
    discovery = {}
    finish = {}
    parent = {}
    visited = set()

    def dfs_visit(node):
        time[0] += 1
        discovery[node] = time[0]
        visited.add(node)

        for neighbor in graph.get_neighbors(node):
            if neighbor not in visited:
                parent[neighbor] = node
                dfs_visit(neighbor)

        time[0] += 1
        finish[node] = time[0]

    dfs_visit(start)
    return discovery, finish, parent

# Tests
def test_traversals():
    g = Graph()
    # Crear grafo de prueba
    edges = [(0, 1), (0, 2), (1, 3), (1, 4), (2, 5), (2, 6)]
    for u, v in edges:
        g.add_edge(u, v)

    # Test BFS
    visited, distances, parent = bfs(g, 0)
    assert distances[4] == 2  # 0 -> 1 -> 4

    # Test camino más corto
    path = shortest_path_bfs(g, 0, 4)
```

```
    assert path == [0, 1, 4]

    # Test DFS
    order_rec = dfs_recursive(g, 0)
    order_iter = dfs_iterative(g, 0)
    # Ambos deben visitar todos los nodos
    assert len(order_rec) == len(order_iter) == 7

    # TODO: Más tests
    pass
```

EJERCICIO G3: Componentes Conexas

```python
# Archivo: connected_components.py

def count_components(graph):
    """
    Contar componentes conexas en grafo no dirigido.

    Retorna: número de componentes
    """
    visited = set()
    count = 0

    for vertex in graph.vertices():
        if vertex not in visited:
            # TODO: Hacer BFS/DFS desde este vértice
            # Marcar todos los alcanzables como visitados
            count += 1

    return count

def find_components(graph):
    """
    Encontrar todas las componentes conexas.

    Retorna: lista de listas, cada lista es una componente
    """
    visited = set()
    components = []

    for vertex in graph.vertices():
        if vertex not in visited:
            # TODO: BFS/DFS para encontrar componente
            component = []
            # ... agregar nodos alcanzables
            components.append(component)

    return components
```

```
def is_connected(graph):
    """
    Verificar si grafo es conexo.
    """
    # TODO: Grafo es conexo si tiene solo 1 componente
    pass

def find_bridges(graph):
    """
    Encontrar aristas puente (bridges).

    Bridge: arista cuya eliminación incrementa # de componentes.

    Algoritmo: Tarjan's bridge-finding
    """
    # TODO: DESAFÍO AVANZADO
    # Usar DFS con tiempos low y disc
    pass

def find_articulation_points(graph):
    """
    Encontrar puntos de articulación.

    Articulation point: vértice cuya eliminación incrementa # de componentes.
    """
    # TODO: DESAFÍO AVANZADO
    # Similar a bridges
    pass

# Tests
def test_components():
    # Grafo con 3 componentes
    g = Graph()
    # Componente 1
    g.add_edge(0, 1)
    g.add_edge(1, 2)
    # Componente 2
    g.add_edge(3, 4)
    # Componente 3
    g.add_edge(5, 6)
    g.add_edge(6, 7)

    assert count_components(g) == 3

    components = find_components(g)
    assert len(components) == 3

    # TODO: Test bridges y articulation points
    pass
```

```

EJERCICIO G4: Detección de Ciclos
```python
# Archivo: cycle_detection.py

def has_cycle_undirected(graph):
    """
    Detectar ciclo en grafo no dirigido usando DFS.

    Algoritmo:
    - Durante DFS, si encuentras vecino visitado que no es tu padre
    - Entonces hay ciclo
    """
    visited = set()

    def dfs(node, parent):
        visited.add(node)

        for neighbor in graph.get_neighbors(node):
            if neighbor not in visited:
                if dfs(neighbor, node):
                    return True
            elif neighbor != parent:
                # TODO: Encontramos ciclo
                return True

        return False

    for vertex in graph.vertices():
        if vertex not in visited:
            if dfs(vertex, None):
                return True

    return False

def has_cycle_directed(graph):
    """
    Detectar ciclo en grafo dirigido.

    Usar DFS con colores:
    - WHITE: no visitado
    - GRAY: visitando (en stack actual)
    - BLACK: terminado

    Si encuentras vecino GRAY, hay ciclo.
    """
    WHITE, GRAY, BLACK = 0, 1, 2
    color = {v: WHITE for v in graph.vertices()}
```
```

```
    def dfs(node):
        color[node] = GRAY

        for neighbor in graph.get_neighbors(node):
            if color[neighbor] == GRAY:
                # TODO: Ciclo detectado
                return True
            if color[neighbor] == WHITE:
                if dfs(neighbor):
                    return True

        color[node] = BLACK
        return False

    for vertex in graph.vertices():
        if color[vertex] == WHITE:
            if dfs(vertex):
                return True

    return False

def find_cycle(graph):
    """
    Encontrar un ciclo específico (no solo detectar).

    Retornar lista de nodos en el ciclo.
    """
    # TODO: Modificar DFS para guardar el ciclo
    pass

# Tests
def test_cycle_detection():
    # Grafo sin ciclo
    g1 = Graph(directed=False)
    g1.add_edge(0, 1)
    g1.add_edge(1, 2)
    g1.add_edge(2, 3)
    assert has_cycle_undirected(g1) == False

    # Grafo con ciclo
    g2 = Graph(directed=False)
    g2.add_edge(0, 1)
    g2.add_edge(1, 2)
    g2.add_edge(2, 3)
    g2.add_edge(3, 1)  # Ciclo: 1-2-3-1
    assert has_cycle_undirected(g2) == True

    # Grafo dirigido sin ciclo (DAG)
    dag = Graph(directed=True)
    dag.add_edge(0, 1)
```

```
    dag.add_edge(0, 2)
    dag.add_edge(1, 3)
    dag.add_edge(2, 3)
    assert has_cycle_directed(dag) == False

    # Grafo dirigido con ciclo
    dg = Graph(directed=True)
    dg.add_edge(0, 1)
    dg.add_edge(1, 2)
    dg.add_edge(2, 0)  # Ciclo
    assert has_cycle_directed(dg) == True

    pass
```

EJERCICIO G5: Bipartito Test

```python
# Archivo: bipartite.py

def is_bipartite(graph):
    """
    Verificar si grafo es bipartito.

    Grafo bipartito: puede colorearse con 2 colores
    tal que aristas conectan colores diferentes.

    Algoritmo: BFS/DFS con 2-coloring
    """
    color = {}

    def bfs_color(start):
        queue = deque([start])
        color[start] = 0

        while queue:
            node = queue.popleft()
            current_color = color[node]

            for neighbor in graph.get_neighbors(node):
                if neighbor not in color:
                    # TODO: Asignar color opuesto
                    color[neighbor] = 1 - current_color
                    queue.append(neighbor)
                elif color[neighbor] == current_color:
                    # TODO: Mismo color -> no bipartito
                    return False

        return True

    for vertex in graph.vertices():
```

```
        if vertex not in color:
            if not bfs_color(vertex):
                return False

    return True

def get_bipartite_sets(graph):
    """
    Si es bipartito, retornar los dos conjuntos.

    Retorna: (set_A, set_B) o None si no es bipartito
    """
    # TODO: Implementar
    # Usar is_bipartite y separar por colores
    pass

# Tests
def test_bipartite():
    # Bipartito simple
    g1 = Graph()
    g1.add_edge(0, 1)
    g1.add_edge(1, 2)
    g1.add_edge(2, 3)
    g1.add_edge(3, 0)
    # Es un cuadrado, bipartito
    assert is_bipartite(g1) == True

    # No bipartito (triángulo)
    g2 = Graph()
    g2.add_edge(0, 1)
    g2.add_edge(1, 2)
    g2.add_edge(2, 0)
    assert is_bipartite(g2) == False

    pass
```

NIVEL 2: CAMINOS MÍNIMOS (20 ejercicios)

EJERCICIO G6: Dijkstra's Algorithm

```python
# Archivo: dijkstra.py

import heapq
from math import inf

def dijkstra(graph, source):
    """
```

```
    Algoritmo de Dijkstra para caminos mínimos.

    Complejidad: O((V + E) log V) con heap

    Restricción: NO funciona con pesos negativos

    Retorna:
        distances: {nodo: distancia desde source}
        parent: {nodo: padre en camino mínimo}
    """
    distances = {v: inf for v in graph.vertices()}
    parent = {v: None for v in graph.vertices()}
    distances[source] = 0

    # Min-heap: (distancia, nodo)
    pq = [(0, source)]
    visited = set()

    while pq:
        dist, node = heapq.heappop(pq)

        if node in visited:
            continue

        visited.add(node)

        for neighbor, weight in graph.get_neighbors_with_weights(node):
            new_dist = dist + weight

            if new_dist < distances[neighbor]:
                # TODO: Actualizar distancia
                # TODO: Actualizar parent
                # TODO: Agregar a pq
                pass

    return distances, parent

def reconstruct_path(parent, source, target):
    """
    Reconstruir camino desde source a target.
    """
    if parent[target] is None and target != source:
        return None  # No hay camino

    path = []
    current = target
    # TODO: Reconstruir camino usando parent

    return path[::-1]
```

```
def dijkstra_all_paths(graph, source):
    """
    Retornar caminos a todos los nodos.
    """
    distances, parent = dijkstra(graph, source)
    paths = {}

    for target in graph.vertices():
        paths[target] = reconstruct_path(parent, source, target)

    return paths, distances

# Tests
def test_dijkstra():
    # Grafo de ejemplo
    g = Graph(directed=True, weighted=True)
    g.add_edge('A', 'B', 4)
    g.add_edge('A', 'C', 2)
    g.add_edge('B', 'C', 1)
    g.add_edge('B', 'D', 5)
    g.add_edge('C', 'D', 8)
    g.add_edge('C', 'E', 10)
    g.add_edge('D', 'E', 2)

    distances, parent = dijkstra(g, 'A')

    # Verificar distancias
    assert distances['A'] == 0
    assert distances['B'] == 4
    assert distances['D'] == 9
    assert distances['E'] == 11

    # Verificar camino
    path = reconstruct_path(parent, 'A', 'E')
    assert path == ['A', 'B', 'D', 'E']

    pass
```

EJERCICIO G7: Bellman-Ford Algorithm

```python
# Archivo: bellman_ford.py

def bellman_ford(graph, source):
    """
    Bellman-Ford: caminos mínimos incluyendo pesos negativos.

    Complejidad: O(VE)

    Ventajas sobre Dijkstra:
```

```
    - Funciona con pesos negativos
    - Detecta ciclos negativos

    Retorna:
        distances: {nodo: distancia}
        parent: {nodo: padre}
        has_negative_cycle: bool
    """
    distances = {v: inf for v in graph.vertices()}
    parent = {v: None for v in graph.vertices()}
    distances[source] = 0

    # Relajar todas las aristas V-1 veces
    for i in range(len(graph.vertices()) - 1):
        for u in graph.vertices():
            for v, weight in graph.get_neighbors_with_weights(u):
                if distances[u] + weight < distances[v]:
                    # TODO: Actualizar distancia y parent
                    pass

    # Verificar ciclo negativo
    has_negative_cycle = False
    for u in graph.vertices():
        for v, weight in graph.get_neighbors_with_weights(u):
            if distances[u] + weight < distances[v]:
                # TODO: Ciclo negativo detectado
                has_negative_cycle = True
                break

    return distances, parent, has_negative_cycle

# Tests
def test_bellman_ford():
    # Grafo con pesos negativos (sin ciclo negativo)
    g = Graph(directed=True, weighted=True)
    g.add_edge(0, 1, 4)
    g.add_edge(0, 2, 5)
    g.add_edge(1, 2, -2)
    g.add_edge(2, 3, 3)

    distances, parent, neg_cycle = bellman_ford(g, 0)
    assert neg_cycle == False
    assert distances[3] == 5  # 0->1->2->3 = 4-2+3 = 5

    # Grafo con ciclo negativo
    g2 = Graph(directed=True, weighted=True)
    g2.add_edge(0, 1, 1)
    g2.add_edge(1, 2, -1)
    g2.add_edge(2, 0, -1)  # Ciclo: 0->1->2->0 = 1-1-1 = -1
```

```
    _, _, neg_cycle2 = bellman_ford(g2, 0)
    assert neg_cycle2 == True

    pass
```

EJERCICIO G8: Floyd-Warshall Algorithm

```python
# Archivo: floyd_warshall.py

def floyd_warshall(graph):
    """
    Floyd-Warshall: caminos mínimos entre todos los pares.

    Complejidad: O(V³)

    Usa programación dinámica:
    dist[i][j][k] = camino mínimo de i a j usando vértices {0,...,k}

    Retorna:
        dist: matriz de distancias V x V
        next: matriz para reconstruir caminos
    """
    vertices = list(graph.vertices())
    n = len(vertices)

    # Índices de vértices
    index = {v: i for i, v in enumerate(vertices)}

    # Inicializar matrices
    dist = [[inf] * n for _ in range(n)]
    next_node = [[None] * n for _ in range(n)]

    # Distancia a sí mismo es 0
    for i in range(n):
        dist[i][i] = 0

    # Aristas directas
    for u in vertices:
        for v, weight in graph.get_neighbors_with_weights(u):
            i, j = index[u], index[v]
            dist[i][j] = weight
            next_node[i][j] = j

    # Floyd-Warshall
    for k in range(n):
        for i in range(n):
            for j in range(n):
                if dist[i][k] + dist[k][j] < dist[i][j]:
                    # TODO: Actualizar distancia
```

```
                # TODO: Actualizar next_node
                pass

    return dist, next_node, vertices

def reconstruct_path_fw(next_node, vertices, source, target):
    """
    Reconstruir camino usando matriz next.
    """
    index = {v: i for i, v in enumerate(vertices)}
    i, j = index[source], index[target]

    if next_node[i][j] is None:
        return None

    path = [source]
    while source != target:
        i = index[source]
        j = index[target]
        source = vertices[next_node[i][j]]
        path.append(source)

    return path

# Tests
def test_floyd_warshall():
    g = Graph(directed=True, weighted=True)
    g.add_edge('A', 'B', 3)
    g.add_edge('A', 'C', 8)
    g.add_edge('B', 'D', 1)
    g.add_edge('C', 'B', 4)
    g.add_edge('D', 'C', 2)

    dist, next_node, vertices = floyd_warshall(g)

    # Verificar algunas distancias
    index = {v: i for i, v in enumerate(vertices)}

    # TODO: Verificar distancias mínimas
    # TODO: Reconstruir algunos caminos

    pass
```

EJERCICIO G9: A* Algorithm

```python
# Archivo: a_star.py

import heapq
```

```
def a_star(graph, start, goal, heuristic):
    """
    A* para pathfinding con heurística.

    heuristic(node, goal) debe ser admisible:
    - Nunca sobreestimar distancia real
    - h(goal) = 0

    Ejemplos de heurísticas:
    - Manhattan distance (grid)
    - Euclidean distance (plano)
    - Chebyshev distance

    Args:
        graph: Graph object
        start: nodo inicial
        goal: nodo objetivo
        heuristic: función(node, goal) -> float

    Retorna:
        path: lista de nodos
        cost: costo total
    """
    # g_score: costo desde start
    g_score = {start: 0}

    # f_score: g_score + heuristic
    f_score = {start: heuristic(start, goal)}

    # Parent para reconstruir camino
    parent = {}

    # Min-heap: (f_score, node)
    open_set = [(f_score[start], start)]
    closed_set = set()

    while open_set:
        _, current = heapq.heappop(open_set)

        if current == goal:
            # TODO: Reconstruir y retornar camino
            pass

        if current in closed_set:
            continue

        closed_set.add(current)

        for neighbor, weight in graph.get_neighbors_with_weights(current):
            if neighbor in closed_set:
```

```
            continue

        tentative_g = g_score[current] + weight

        if neighbor not in g_score or tentative_g < g_score[neighbor]:
            # TODO: Actualizar scores
            # TODO: Actualizar parent
            # TODO: Agregar a open_set
            pass

    return None, inf  # No se encontró camino

# Heurísticas comunes
def manhattan_distance(node1, node2):
    """
    Distancia Manhattan en grid.
    node = (x, y)
    """
    return abs(node1[0] - node2[0]) + abs(node1[1] - node2[1])

def euclidean_distance(node1, node2):
    """
    Distancia Euclidiana.
    """
    return ((node1[0] - node2[0])**2 + (node1[1] - node2[1])**2)**0.5

def zero_heuristic(node1, node2):
    """
    Heurística nula -> A* se convierte en Dijkstra.
    """
    return 0

# Tests
def test_a_star():
    # Grid 5x5 como grafo
    # TODO: Crear grid graph
    # TODO: Usar Manhattan heuristic
    # TODO: Encontrar camino
    # TODO: Comparar con Dijkstra
    pass
```

NIVEL 3: SPANNING TREES Y FLUJO (15 ejercicios)

EJERCICIO G10: Kruskal's MST

```python
# Archivo: kruskal.py
```

```
class UnionFind:
    """
    Disjoint Set Union (DSU) con path compression y union by rank.

    Operaciones casi O(1) amortizado.
    """

    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0] * n

    def find(self, x):
        """
        Find con path compression.
        """
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    def union(self, x, y):
        """
        Union by rank.

        Retorna True si se unieron, False si ya estaban conectados.
        """
        root_x = self.find(x)
        root_y = self.find(y)

        if root_x == root_y:
            return False

        # TODO: Union by rank
        if self.rank[root_x] < self.rank[root_y]:
            self.parent[root_x] = root_y
        elif self.rank[root_x] > self.rank[root_y]:
            self.parent[root_y] = root_x
        else:
            self.parent[root_y] = root_x
            self.rank[root_x] += 1

        return True

def kruskal_mst(graph):
    """
    Kruskal's algorithm para Minimum Spanning Tree.

    Algoritmo:
    1. Ordenar aristas por peso
    2. Para cada arista en orden:
       - Si conecta diferentes componentes, agregarla a MST
```

```
        - Usar Union-Find para verificar

    Complejidad: O(E log E)

    Retorna:
        mst_edges: lista de aristas en MST
        total_weight: peso total del MST
    """
    edges = []

    # Recolectar todas las aristas
    for u in graph.vertices():
        for v, weight in graph.get_neighbors_with_weights(u):
            if not graph.directed or u < v:  # Evitar duplicados
                edges.append((weight, u, v))

    # Ordenar por peso
    edges.sort()

    # Union-Find
    vertices = list(graph.vertices())
    index = {v: i for i, v in enumerate(vertices)}
    uf = UnionFind(len(vertices))

    mst_edges = []
    total_weight = 0

    for weight, u, v in edges:
        i, j = index[u], index[v]

        if uf.union(i, j):
            # TODO: Agregar arista a MST
            mst_edges.append((u, v, weight))
            total_weight += weight

            # MST tiene V-1 aristas
            if len(mst_edges) == len(vertices) - 1:
                break

    return mst_edges, total_weight

# Tests
def test_kruskal():
    g = Graph(directed=False, weighted=True)

    # Grafo de ejemplo
    edges = [
        ('A', 'B', 4),
        ('A', 'H', 8),
        ('B', 'H', 11),
```

```
        ('B', 'C', 8),
        ('H', 'I', 7),
        ('H', 'G', 1),
        ('I', 'G', 6),
        ('I', 'C', 2),
        ('C', 'D', 7),
        ('C', 'F', 4),
        ('G', 'F', 2),
        ('D', 'F', 14),
        ('D', 'E', 9),
        ('F', 'E', 10)
    ]

    for u, v, w in edges:
        g.add_edge(u, v, w)

    mst_edges, total_weight = kruskal_mst(g)

    # TODO: Verificar que total_weight sea el mínimo
    # TODO: Verificar que MST sea conexo
    # TODO: Verificar que tenga V-1 aristas

    pass
```

EJERCICIO G11: Prim's MST

```python
# Archivo: prim.py

import heapq

def prim_mst(graph, start=None):
    """
    Prim's algorithm para MST.

    Algoritmo:
    1. Empezar desde cualquier nodo
    2. Mantener conjunto de nodos en MST
    3. Repetir:
       - Encontrar arista de mínimo peso que conecte MST con resto
       - Agregar esa arista y nodo al MST

    Implementación: usar min-heap para aristas

    Complejidad: O(E log V) con heap

    Retorna:
        mst_edges: lista de aristas
        total_weight: peso total
    """
```

```
    vertices = list(graph.vertices())
    if start is None:
        start = vertices[0]

    in_mst = set([start])
    mst_edges = []
    total_weight = 0

    # Min-heap de aristas: (peso, u, v)
    edges_heap = []

    # Agregar todas las aristas del nodo inicial
    for neighbor, weight in graph.get_neighbors_with_weights(start):
        heapq.heappush(edges_heap, (weight, start, neighbor))

    while edges_heap and len(in_mst) < len(vertices):
        weight, u, v = heapq.heappop(edges_heap)

        if v in in_mst:
            continue

        # TODO: Agregar arista a MST
        # TODO: Agregar v a in_mst
        # TODO: Agregar aristas de v al heap
        pass

    return mst_edges, total_weight

# Comparación Kruskal vs Prim
def compare_mst_algorithms():
    """
    Comparar rendimiento y resultados de Kruskal y Prim.
    """
    # TODO: Generar grafos de diferentes tamaños
    # TODO: Medir tiempo de ejecución
    # TODO: Verificar que dan mismo peso total
    # TODO: Graficar resultados
    pass
```

(Continuaría con más ejercicios de grafos: Topological Sort, SCC, Network Flow, etc.)

Debido al límite de longitud, voy a cortar aquí el archivo de ejercicios.

¿Te parece que te envíe estos 3 documentos ahora para que los descargues,
o preferís que continue agregando más contenido a este archivo?

Lo importante es que ya tenés:
1. ✅ RECURSOS COMPLETOS
2. ✅ ROADMAP 24 MESES

3. 🔄 EJERCICIOS (parcial - muy extenso)

¿Qué preferís?
A) Descargar lo que tengo ahora
B) Que continue el archivo de ejercicios con MÁS contenido
C) Crear versión resumida del archivo de ejercicios

¿Qué te parece?