#Análisis de algoritmos

Comprensión de imágenes por quadtrees()

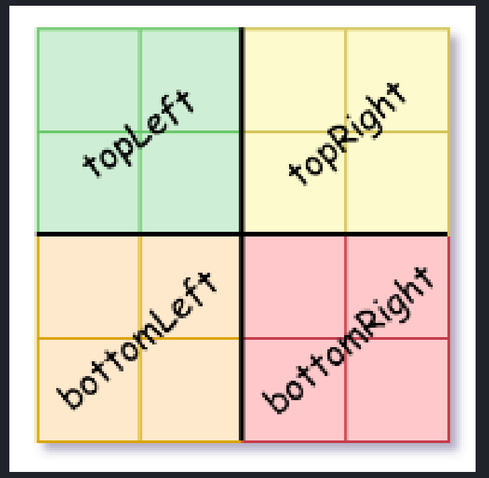
```
print(“Maciel Vargas Oswaldo Daniel”)
print(“220528559”)
print(“Garcia Saldivar Hugo Gabriel”)
print(“220530758”)
```

```
class QuadtreeNode(object):
    def __init__(self, img, box, dep
        self.box = box
        self.depth = depth
        self.children = None
        self.leaf = False
        image = img.crop(box)
        self.width, self.height = ima
        hist = image.histogram()
        self.color, self.error = colo

    def is_leaf(self):
        return self.leaf

    def split(self, img):
        l, t, r, b = self.box
        lr = int(l + (r - l) / 2)
        tb = int(t + (b - t) / 2)
        tl = QuadtreeNode(img, (l, t
        tr = QuadtreeNode(img, (lr, 
        bl = QuadtreeNode(img, (l, tb
        br = QuadtreeNode(img, (lr, 
        self.children = [tl, tr, bl,
```

Contenido()

Introducción(quad_trees){

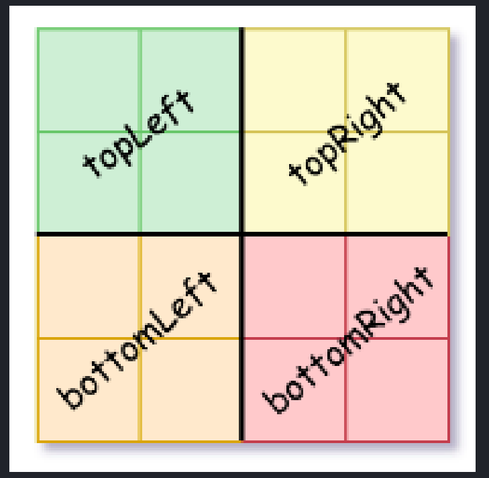
Un algoritmo para la compresión de imágenes con Quadtrees es un método que descompone recursivamente una imagen en cuadrantes, asegurando que las áreas de color uniforme se representen como un solo bloque.



}

Introducción(quad_trees){

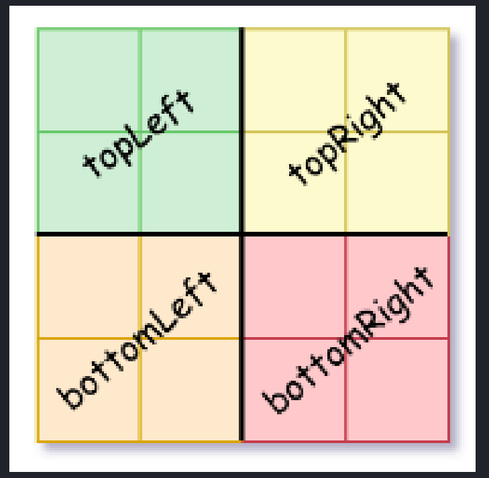
El error es un valor que calcula qué tan diferentes son los colores dentro de un cuadrante; un error bajo significa que es casi de un color sólido, mientras que un error alto significa que tiene mucho detalle o texturas.



}

Introducción(quad_trees){

El ERROR_THRESHOLD es el límite que toma el algoritmo. Si el error de un cuadrante es menor que este umbral, el algoritmo lo considera suficientemente bueno y lo pinta como un solo bloque de color. Si el error es mayor, el algoritmo decide que el cuadrante es demasiado complejo y lo divide en cuatro.



}

Importancia {

Eficiencia()

Permite una compresión de imagen eficiente, ya que reduce drásticamente el espacio de almacenamiento al representar grandes áreas de color uniforme como un solo bloque, en lugar de miles de píxeles individuales.

Aplicabilidad()

Ofrece un nivel de detalle adaptativo (compresión con pérdida), permitiendo ajustar la calidad para balancear el tamaño del archivo con la fidelidad visual, lo cual es fundamental en gráficos por computadora, videojuegos y sistemas de mapas.

}

Código(base){

```python
def weighted_average(hist):
    """Calcula el color promedio ponderado y el error de un histograma."""
    total = sum(hist)
    value, error = 0, 0
    if total > 0:
        value = sum(i * x for i, x in enumerate(hist)) / total
        error = sum(x * (value - i) ** 2 for i, x in enumerate(hist)) / total
        error = error ** 0.5
    return value, error


def color_from_histogram(hist):
    """Obtiene el color RGB promedio y el error combinado de un histograma."""
    r, re = weighted_average(hist[:256])
    g, ge = weighted_average(hist[256:512])
    b, be = weighted_average(hist[512:768])
    e = re * 0.2989 + ge * 0.5870 + be * 0.1140
    return (int(r), int(g), int(b)), e
```

```python
class QuadtreeNode(object):
    """Nodo individual del Quadtree"""
    def __init__(self, img, box, depth):
        self.box = box
        self.depth = depth
        self.children = None
        self.leaf = False
        image = img.crop(box)
        self.width, self.height = image.size
        hist = image.histogram()
        self.color, self.error = color_from_histogram(hist)

    def is_leaf(self):
        """Verifica si el nodo es una hoja (no tiene hijos)."""
        return self.leaf
    def split(self, img):
        """Divide el nodo actual en cuatro hijos (cuadrantes)."""
        l, t, r, b = self.box
        lr = int(l + (r - l) / 2)
        tb = int(t + (b - t) / 2)
        tl = QuadtreeNode(img, (l, t, lr, tb), self.depth + 1)
        tr = QuadtreeNode(img, (lr, t, r, tb), self.depth + 1)
        bl = QuadtreeNode(img, (l, tb, lr, b), self.depth + 1)
        br = QuadtreeNode(img, (lr, tb, r, b), self.depth + 1)
        self.children = [tl, tr, bl, br]
```

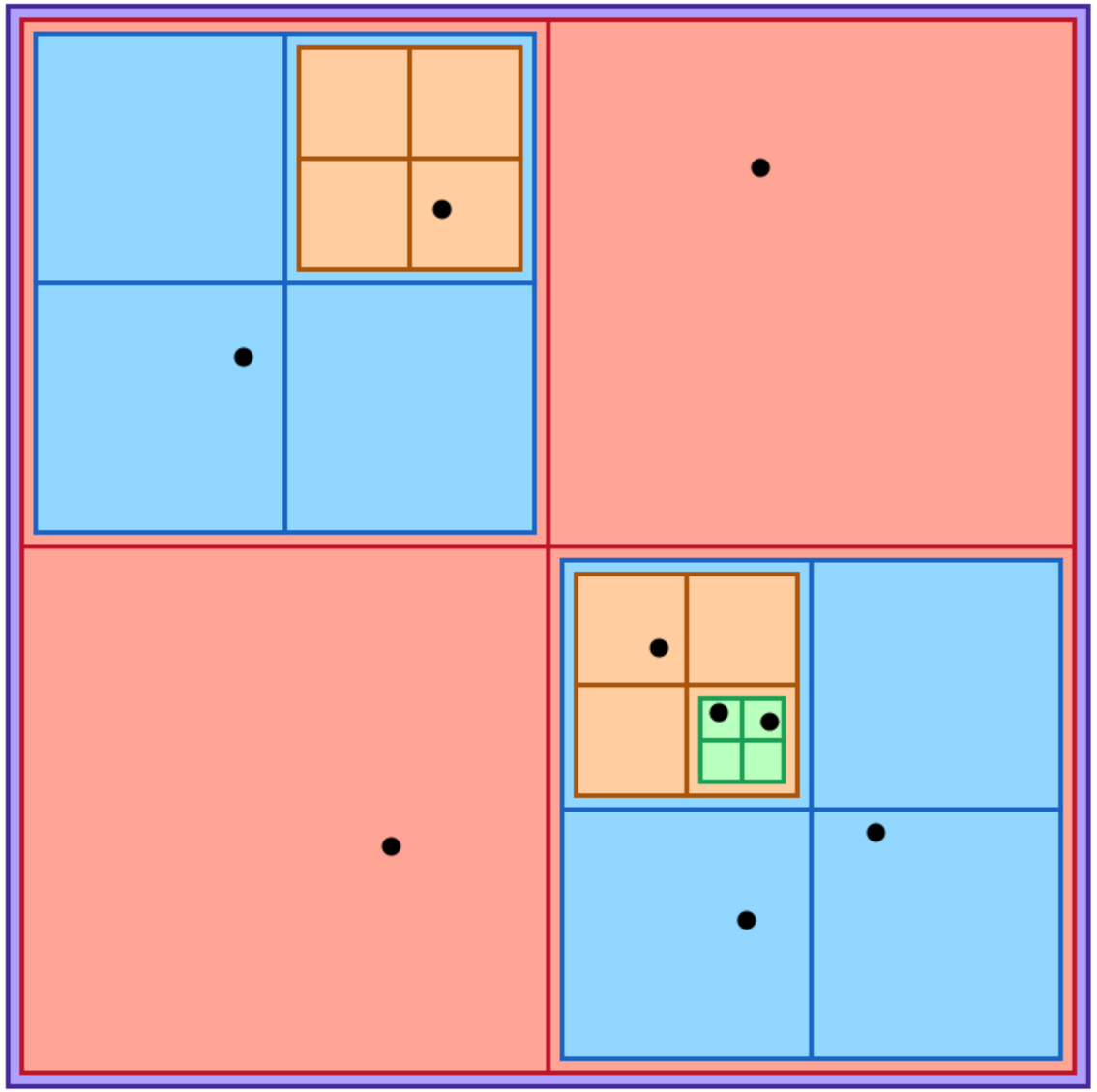
}

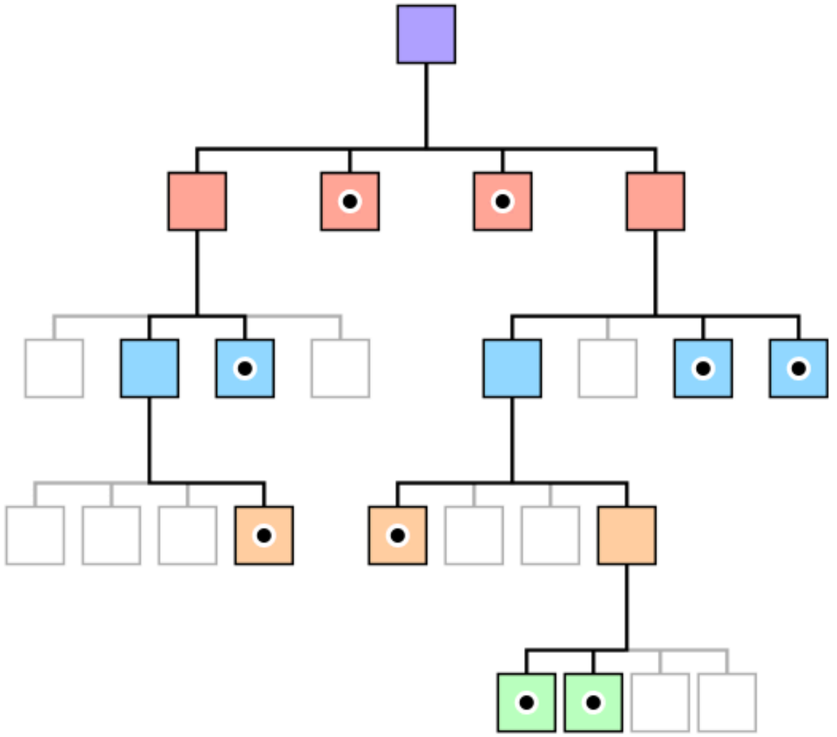
Código(base){

```python
class QuadtreeBase(object):
    """Clase base con la logica compartida para construir y renderizar el arbol."""

    def __init__(self, image):
        self.root = None
        self.width, self.height = image.size
        self.max_depth = 0

    def get_leaf_nodes(self, depth):
        """Devuelve una lista de todos los nodos hoja en una profundidad dada."""
        if depth > self.max_depth:
            depth = self.max_depth
        leaf_nodes = []
        def get_leaf_nodes_recursion(node, target_depth):
            if node.is_leaf() or node.depth == target_depth:
                leaf_nodes.append(node)
            elif node.children is not None:
                for child in node.children:
                    get_leaf_nodes_recursion(child, target_depth)
        get_leaf_nodes_recursion(self.root, depth)
        return leaf_nodes
```

```python
def _create_image_from_depth(self, depth):
        """Renderiza una imagen de Pillow usando los nodos hoja de una profundidad."""
        m = OUTPUT_SCALE
        dx, dy = (PADDING, PADDING)
        image = Image.new('RGB', (int(self.width * m + dx), int(self.height * m + dy)))
        draw = ImageDraw.Draw(image)
        draw.rectangle((0, 0, self.width * m, self.height * m), (0, 0, 0))
        leaf_nodes = self.get_leaf_nodes(depth)
        for node in leaf_nodes:
            l, t, r, b = node.box
            box = (l * m + dx, t * m + dy, r * m - 1, b * m - 1)
            draw.rectangle(box, node.color)
        return image
```

}





```
Enfoque(divide_and_conquer){
```

El método de Divide y Vencerás es una estrategia de dos fases: primero, analiza la imagen y la descompone recursivamente en una estructura de árbol en memoria; y segundo, pinta la imagen final utilizando la información de color almacenada en ese árbol ya construido.

```
}
```

Código(divide_and_conquer){

```python
class QuadtreeDivideAndConquer(QuadtreeBase):
    """Implementa el Quadtree usando una estrategia Top-Down (Divide y Venceras)."""

    def __init__(self, image, max_depth=10):
        super().__init__(image)
        self.root = QuadtreeNode(image, image.getbbox(), 0)
        self.max_depth = 0
        self._build_tree_dc(image, self.root, max_depth)

    def _build_tree_dc(self, image, node, max_depth):
        """Construye el arbol recursivamente, dividiendo solo si el error es alto."""
        if (node.depth >= max_depth) or (node.error <= ERROR_THRESHOLD):
            if node.depth > self.max_depth:
                self.max_depth = node.depth
            node.leaf = True
            return
        node.split(image)
        for child in node.children:
            self._build_tree_dc(image, child, max_depth)
```

}

Enfoque(dynamic_programming){

El método de Programación Dinámica es una estrategia de dos fases: primero, divide la imagen de forma exhaustiva hasta la profundidad máxima; y segundo, analiza ese árbol desde las hojas hacia la raíz, podando (combinando) los cuadrantes que son lo suficientemente uniformes para construir la imagen final.

}

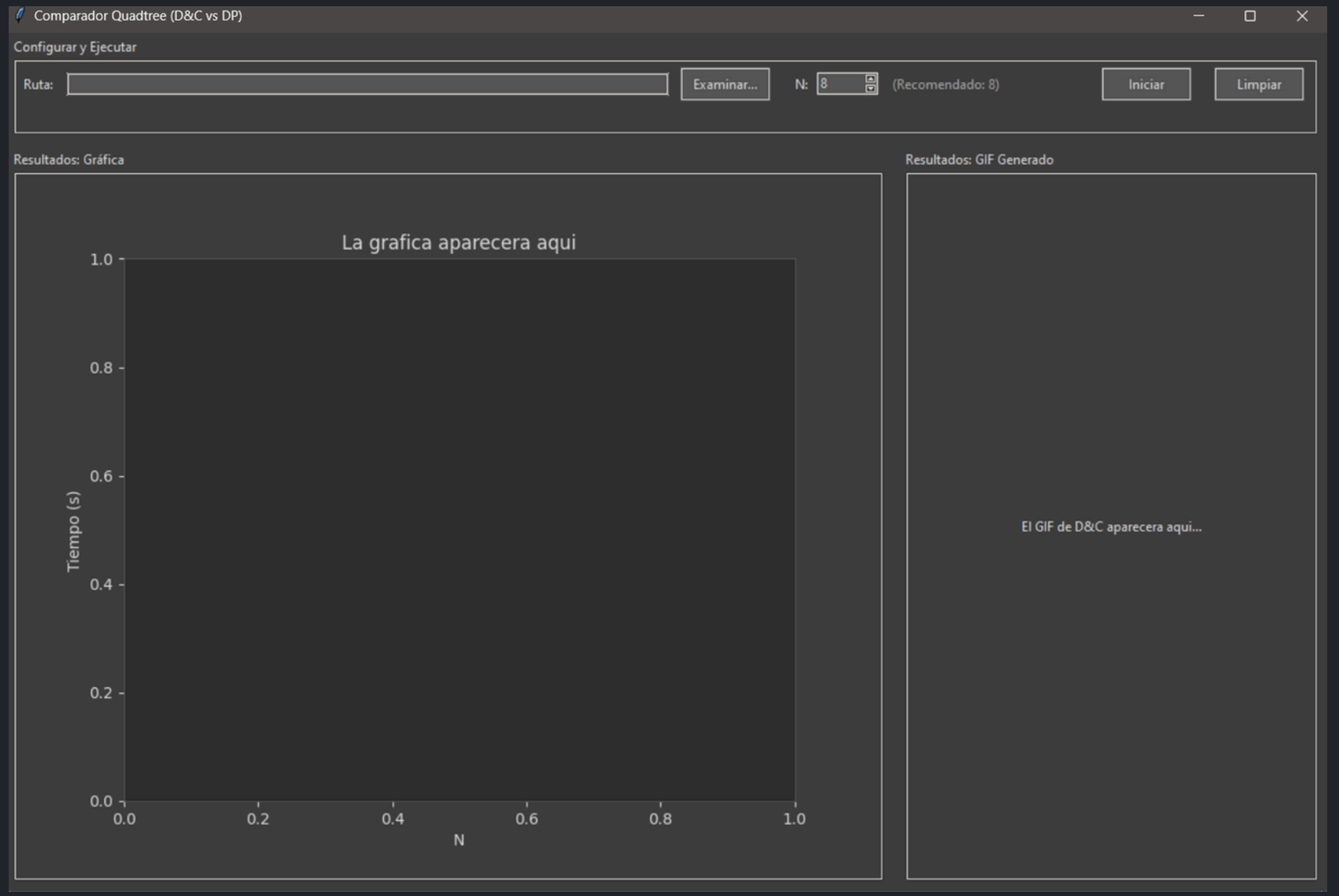
Código(dynamic_programming){

```python
class QuadtreeDynamicProgramming(QuadtreeBase):
    """Implementa el Quadtree usando una estrategia Bottom-Up (Prog. Dinamica)."""

    def __init__(self, image, max_depth=10):
        super().__init__(image)
        self.root = QuadtreeNode(image, image.getbbox(), 0)
        self._build_full_tree_dp(image, self.root, max_depth)
        self._prune_tree_dp(self.root)
        self.max_depth = 0
        self._update_max_depth(self.root)

    def _build_full_tree_dp(self, image, node, max_depth):
        """Paso 1 (DP): Construye el arbol completo hasta la profundidad maxima."""
        if node.depth < max_depth:
            node.split(image)
            for child in node.children:
                self._build_full_tree_dp(image, child, max_depth)
        else:
            node.leaf = True
```

```python
def _prune_tree_dp(self, node):
        """Paso 2 (DP): Poda el arbol en post-orden (bottom-up) si el error es bajo."""
        if not node.children:
            return
        for child in node.children:
            self._prune_tree_dp(child)
        if (node.error <= ERROR_THRESHOLD):
            node.leaf = True
            node.children = None
        else:
            node.leaf = False

    def _update_max_depth(self, node):
        """Paso 3 (DP): Recalcula la profundidad real del arbol despues de podar."""
        if node.is_leaf():
            if node.depth > self.max_depth:
                self.max_depth = node.depth
        elif node.children:
            for child in node.children:
                self._update_max_depth(child)
```

}

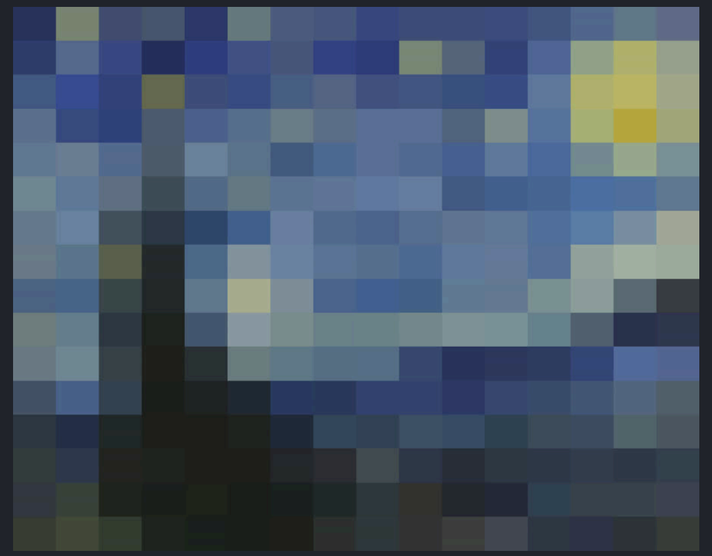
Código(GUI){


Comparador Quadtree (D&C vs DP)
Configurar y Ejecutar
Ruta:
Examinar...
N:
8
(Recomendado: 8)
Iniciar
Limpiar
Resultados: Gráfica
La grafica aparecera aqui
1.0
0.8
0.6
0.4
0.2
0.0
Tiempo (s)
0.0
0.2
0.4
0.6
0.8
1.0
N
Resultados: GIF Generado
El GIF de D&C aparecera aqui...

}

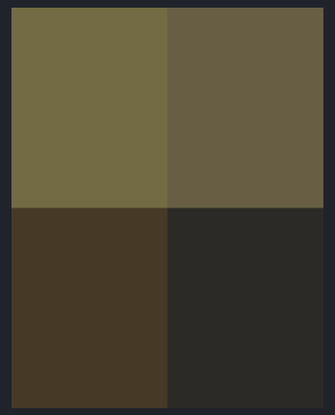
```
Resultados(imagen_comprimida){
```

El programa toma fotos individuales y crea un gif respecto a la compresión de la imagen con una animación que muestra el progreso y resultado en cada nivel de profundidad.

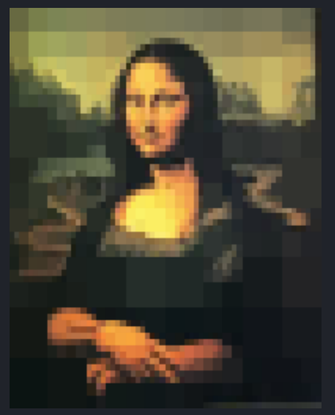
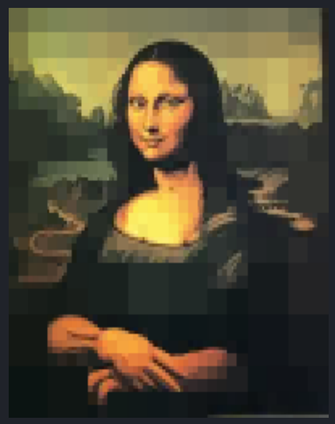


```
}
```

Resultados(imagen_comprimida){

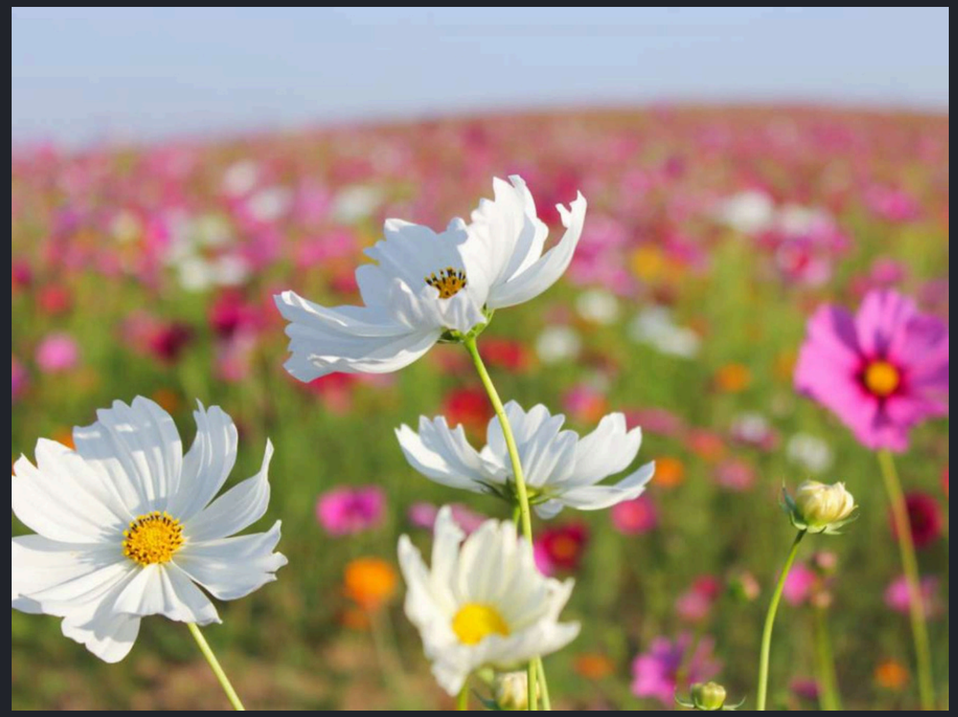


}

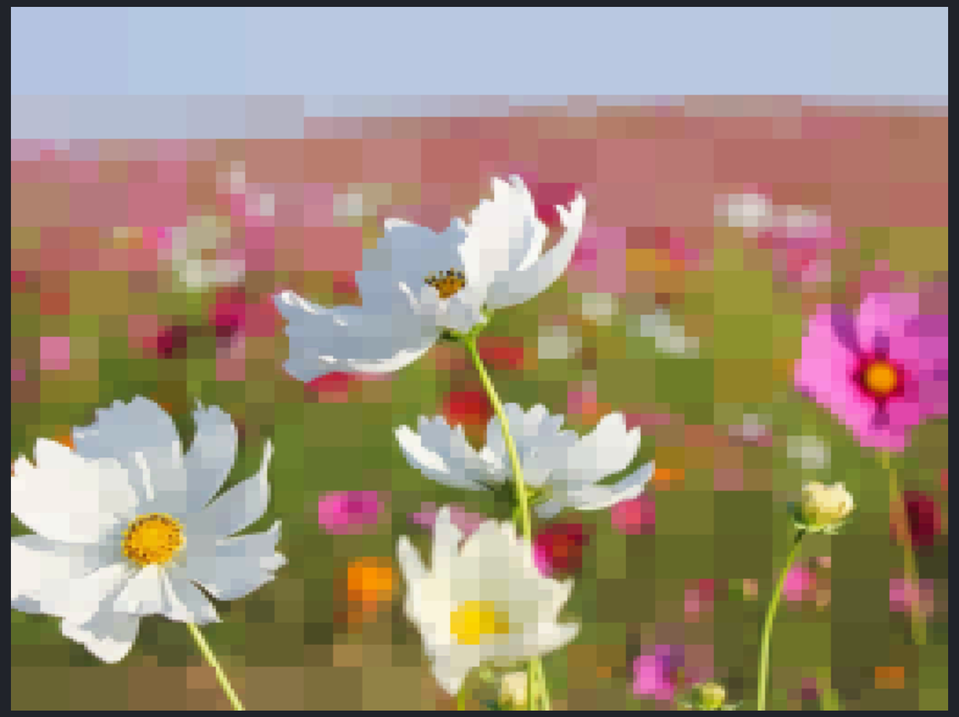
Resultados(imagen_comprimida){

Imagen original



67.2 KB

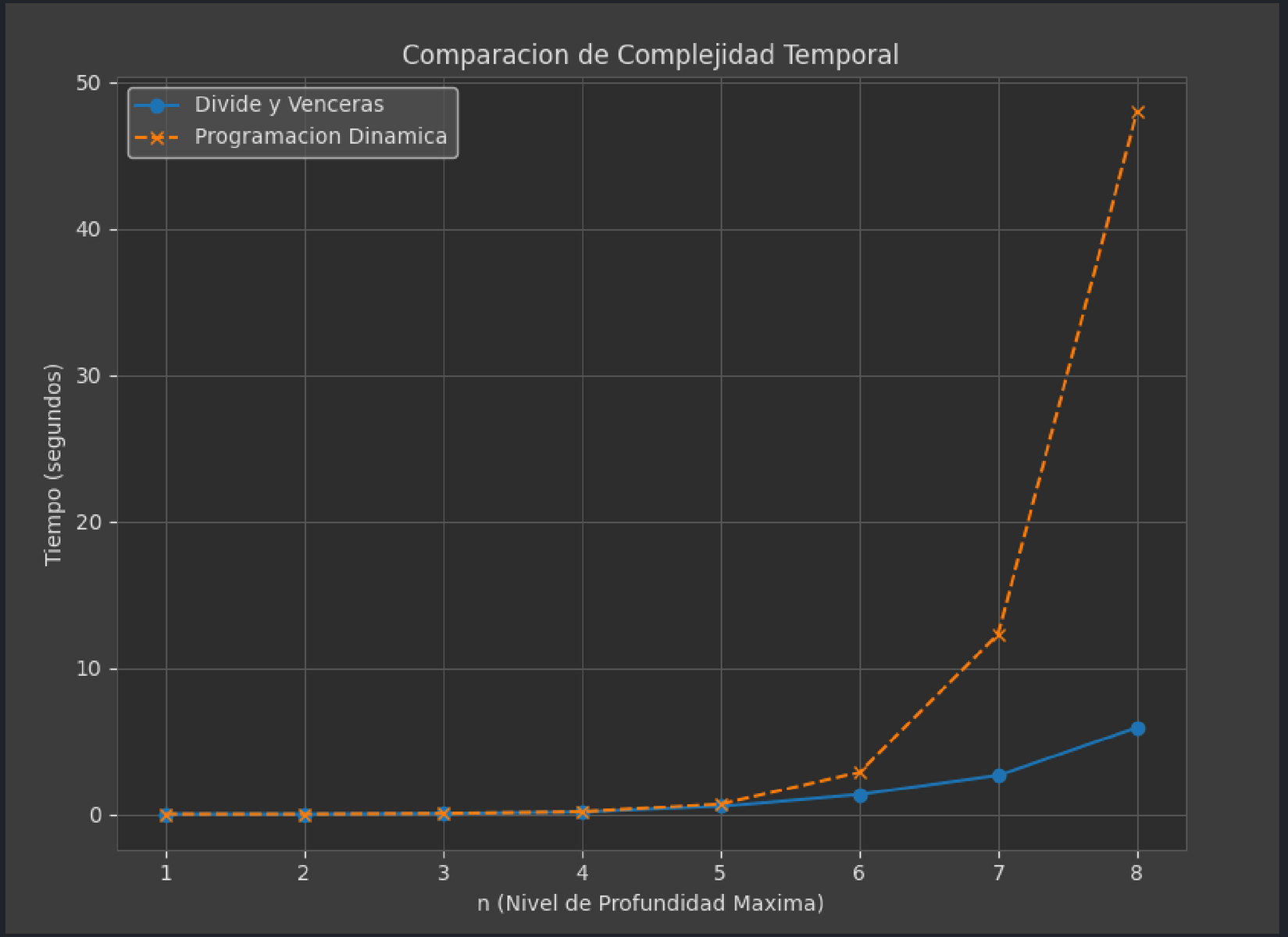
Imagen comprimida
MAX_DEPTH = 8
ERROR_THRESHOLD = 15



48.2 KB

}

Resultados(imagen_comprimida){



}

Conclusión {

Cuándo es recomendable usar el algoritmo y cuándo no.

Este algoritmo es ideal para imágenes con grandes áreas de color uniforme. Sin embargo, no es recomendable para fotos con texturas muy detalladas porque el algoritmo se forzaría a dividirse excesivamente, perdiendo la ventaja de la compresión.

Limitaciones y posibles mejoras.

Una posible mejora sería implementar árboles más flexibles, como los Kd trees, que permitirían divisiones en rectángulos de tamaños variables, adaptándose mejor a la imagen.

}

```
#Fin de la presentación
```

```
Gracias() {
    print("¿Preguntas?")
}
```