ESCUELA POLITÉCNICA NACIONAL
ESCUELA DE FORMACIÓN DE TECNÓLOGOS

ALGORITMOS Y ESTRUCTURAS DE DATOS

ASIGNATURA:	Algoritmos y Estructuras de Datos
PROFESOR:	Ing. Lorena Chulde
FECHA:	04 – 02 - 2026
PERÍODO ACADÉMICO:	2025-B

PROYECTO FINAL

Nombres de los estudiantes:

Leonardo Defaz
Alexis Chasi
Ángel Mena
Jhon Freire

Índice de Contenido

Índice de Tablas

Índice de Ilustraciones

1. Introducción:

Hoy en día, todas las empresas que se dedican al envío de paquetes tienen el reto enorme de encontrar la mejor forma de mover sus productos entre sus bodegas y los puntos donde los clientes esperan, lo cual no es nada fácil en un mercado donde cada segundo y cada centavo cuentan. Esta dificultad no se trata solamente de gastar menos dinero o recorrer menos kilómetros, sino de lograr que el servicio sea tan rápido y bueno que la gente prefiera usar esa empresa sobre cualquier otra competencia. Es precisamente en este escenario donde nace el Sistema Inteligente de Distribución para "PoliDelivery", que es un programa diseñado para funcionar de manera ágil y profesional, permitiendo crear un modelo digital de cómo funcionan las redes de transporte y los centros donde se guarda la mercadería.

El problema que queremos resolver con este proyecto es que muchas veces no se sabe con exactitud cuál es el camino más barato o el más corto entre dos ciudades, ni se tiene una forma ordenada de ver cómo están repartidos los centros de distribución por todo el país. Por esta razón, el sistema no se limita a ser una simple herramienta de cálculo, sino que se convierte en la prueba real de que podemos usar todo lo aprendido en nuestras clases sobre cómo organizar la información y cómo crear soluciones lógicas para problemas que ocurren en la vida diaria de cualquier negocio.

Para que el programa sea realmente fuerte y confiable, hemos utilizado formas avanzadas de guardar la información, como los grafos que sirven para dibujar las conexiones entre ciudades y los árboles que permiten separar los centros por regiones de manera ordenada. El sistema utiliza fórmulas matemáticas y lógicas muy famosas, como el algoritmo de Dijkstra que sirve para encontrar la ruta que menos golpea el bolsillo de la empresa, y también otros métodos que exploran todas las opciones posibles de entrega nivel por nivel. Al final, para que nada de esto se pierda, el programa guarda todo en archivos de texto simples que permiten manejar las cuentas de los usuarios y crear rutas que se ajusten a lo que cada cliente necesita, logrando así una herramienta que es fácil de usar pero que por dentro tiene una ingeniería muy completa y profesional.

2. Objetivos:

Desarrollar un proyecto funcional para la empresa "PoliDelivery" que resuelva situaciones logísticas reales mediante la aplicación práctica de estructuras de datos y lógica algorítmica para la optimización de procesos.

- Optimizar la planificación de rutas de entrega y reducir los costos operativos mediante la implementación de algoritmos clásicos de grafos como Dijkstra, BFS y DFS.
- Organizar y gestionar la información de los centros logísticos de manera jerárquica y eficiente utilizando árboles para regiones y estructuras de datos como listas, diccionarios y matrices de costos.
- Mejorar la experiencia del usuario (administrador y cliente) a través de un sistema que permita la búsqueda avanzada, el ordenamiento de datos y el almacenamiento persistente de rutas en archivos externos.

3. Descripción Del Sistema:

El sistema “Polidelivery” es una aplicación desarrollada en Python que funciona mediante consola y su objetivo es optimizar la distribución de paquetes entre centros de distribución, todo esto haciendo uso de algritmos y estructuras de datos revisadas en clase.

El sistema permite modelar los centros de distribución como nodos dentro deun grafo donde las rutas contienen información de distancia y costo. A partir de las distribuciones de los centros el sistema es capaz de calcular rutas óptimas y explorar conexiones entre centros. De igual manera el sistema permite gestionar un registro y autenticación de usuarios, diferenciado roles entre administrador y usuario, utilizando archivos (.txt.)

Roles del Sistema:

1. **Administrador:** Tiene control total sobre la gestión del sistema, permitiendo visualizar listas de centros distribución, agregar, actualizar o eliminar centros de distribución, gestionar rutas y guardar cambios.

2. **Cliente:** El rol cliente está orientado a la consulta y exploración de las rutas de distribución y sus costos. Permite visualizar el mapa de centros, Calcular la ruta más económica mediante el algoritmo de Dijkstra, explorar centros cercanos por BFS y recorrer redes por DFS.

Alcance del sistema:

El sistema de "Polidelivery" en consola se limita a la simulación de un sistema de distribución de paquetes sin interfaz gráfica, per siendo capaz de implementar estructuras de datos como listas, diccionarios, grafos entre otros, es capaz de aplicar algoritmos de ordenamiento, búsqueda y otros como Dijkstra, BFS Y DFS, maneja información de manera persistente gracias al uso de archivos de texto. Finalmente es capaz de interactuar con el usuario según su rol.

4. Arquitectura del Sistema:

Tipo de aplicación (Consola)

PoliDelivery es una aplicación de consola programada en Python que permite al usuario interactuar mediante menús de textos. Todas las operaciones del sistema se realizan ingresando las opciones desde el teclado y visualizando los resultados en pantalla.
Al no utilizar una interfaz gráfica, el sistema es ligero, fácil de ejecutar y compatible con cualquier equipo que disponga de Python.

El sistema está desarrollado bajo una arquitectura de archivo único. En este se integran las estructuras de datos, los algoritmos de búsqueda, el manejo de archivos y la lógica de control del sistema.

Diagrama Lógico del Sistema

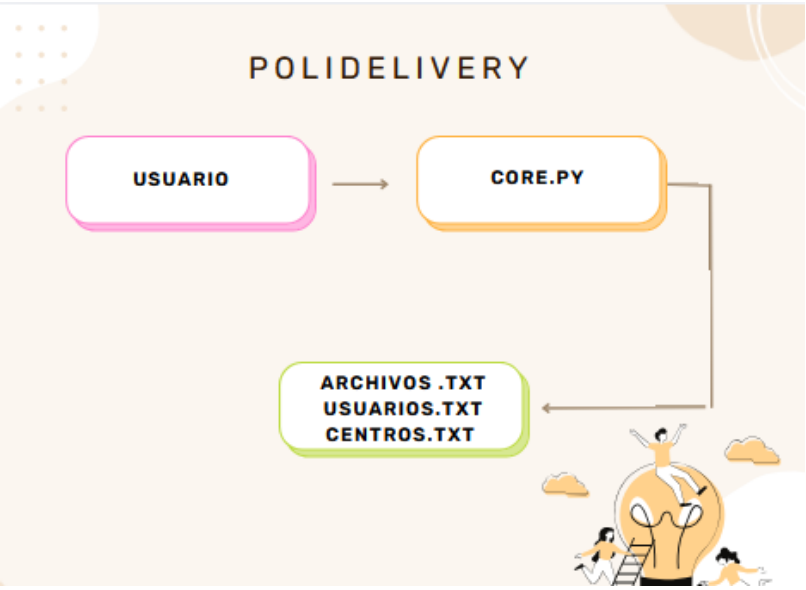


Ilustración 1 Arquitectura del Sistema

Flujo general de ejecución

Al iniciar el sistema "PoliDelivery", se inicializan las variables y estructuras de datos que permiten almacenar y gestionar la información del sistema. El usuario ingresa una opción a través del teclado, la cual es validada por el sistema. En función de la opción seleccionada, el programa realiza las operaciones correspondientes y procesa la información involucrada.

Una vez validada la opción, el sistema ejecuta la funcionalidad correspondiente, procesa la información requerida y muestra los resultados en la consola. Al finalizar cada operación, el programa retorna al menú principal, permitiendo al usuario realizar nuevas acciones hasta que seleccione la opción de salida, momento en el cual el sistema termina su ejecución.

5. Herramientas y Tecnologías Utilizadas:

El lenguaje de programación usado para realizar el sistema "PoliDelivery" fue Python en el entorno de desarrollo Visual Studio Code, recomendado por su integración nativa con las terminales y extensiones que se le pueden agregar de Python.

Se usaron varias librerías de Python que permitieron el desarrollo del sistema como:

1. **Librería OS:** utilizada para las operaciones que involucraban los archivos, como el acceso y manipulación de rutas o directorios.

2. **Librería RE:** se incorporó para el uso de expresiones regulares, facilitando la validación y el procesamiento de datos ingresados por el usuario.

3. **Librería MATH:** permitió realizar cálculos matemáticos requeridos en ciertos procedimientos del sistema, especialmente en operaciones relacionadas con distancias o valores numéricos.

4. **Librería HEAPQ:** fue fundamental para implementar estructuras de tipo cola de prioridad, utilizadas principalmente en algoritmos de rutas óptimas como Dijkstra, mejorando la eficiencia en la búsqueda del camino más corto.

6. Estructuras de Datos Utilizados:

El núcleo del sistema utiliza una arquitectura híbrida de estructuras de datos para organizar la información:

1. **Grafos:** Los centros de distribución se modelan como nodos dentro de un grafo, donde las aristas representan rutas físicas que contienen datos de distancia y costo.

2. **Matrices de Costo:** Se emplea una matriz para consolidar los costos entre centros, facilitando el procesamiento de datos por los algoritmos de búsqueda.

3. **Árboles:** La organización geográfica de la empresa se gestiona mediante un árbol jerárquico que clasifica los centros por regiones y subregiones, permitiendo una exploración ordenada del mapa logístico.

El sistema está equipado con un motor de procesamiento que incluyen:

1. **Optimización de Rutas:** Implementación del algoritmo de Dijkstra para determinar la trayectoria con el costo más bajo entre dos puntos.

2. **Exploración de Redes:** Uso de algoritmos BFS (búsqueda en anchura) para identificar centros cercanos y DFS (búsqueda en profundidad) para la exploración completa de rutas disponibles.

3. **Gestión de Información:** Integración de cinco métodos de ordenamiento (Burbuja, Inserción, Selección, Merge Sort y Quick Sort) y tres métodos de búsqueda (Lineal, Binaria e Interpolación) para la manipulación eficiente de la base de datos.

4. **Colas de Prioridad (Heaps):** Utilizadas para optimizar la ejecución del algoritmo de Dijkstra, asegurando tiempos de respuesta mínimos al calcular la ruta más económica.

7. Algoritmos Implementados:

La información se encuentra en anexos.

8. Gestión de Archivos:

El sistema implementa mecanismos de lectura y escritura de archivos de texto para garantizar la persistencia de los datos. Al iniciar la aplicación, la información almacenada se carga en estructuras de datos internas, y al realizar modificaciones, estas se escriben nuevamente en los archivos correspondientes.

Descripción de archivos utilizados:

- **centros.txt**: archivo encargado de almacenar la información de los centros de distribución y las rutas asociadas. Se encuentra organizado en secciones claramente identificadas para facilitar su lectura y mantenimiento.

```
def leer_centros_rutas():
    centros, rutas = [], []
    mode = None
    with open("centros.txt", "r", encoding="utf-8") as f:
```

Ilustración 2 Función leer_centros_rutas():

```
def guardar_centros_rutas(centros, rutas):
    with open("centros.txt", "w", encoding="utf-8") as f:
        f.write("[CENTROS]\n")
        for c in centros:
            f.write(f"{c.cid}|{c.nombre}|{c.region}|{c.subregion}\n")
        f.write("[RUTAS]\n")
        for r in rutas:
            f.write(f"{r.a}|{r.b}|{r.distancia}|{r.costo}\n")
```

Ilustración 3 Función guardar_centros_rutas():

- **usuarios.txt**: archivo que almacena los datos de autenticación y perfil de los usuarios, permitiendo la validación de credenciales y la gestión de roles

```
def leer_usuarios():
    users = []
    if not os.path.exists("usuarios.txt"):
        return users
    with open("usuarios.txt", "r", encoding="utf-8") as f:
```

Ilustración 4 Función leer_usuarios():

```
def guardar_usuarios(usuarios):
    with open("usuarios.txt", "w", encoding="utf-8") as f:
        for u in usuarios:
            f.write(f"{u['email']}|{u['password']}|{u['nombre']}|{u['identificacion']}|{u['edad']}|{u['rol']}\n")
```

Ilustración 5 Función guardar_usuarios():

9. Validaciones y Seguridad:

Registro de usuarios

Se verifican que los datos ingresados por el usuario durante su registro sean los correctos.

```
=== POLIDELIVERY ===
1. Iniciar sesión
2. Registrarse
3. Salir
Opción: 2
Nombre: alexander
Identificación: 03241132
Edad: 13
Email (@gmail.com): dsf
Contraseña: sdaf
Email inválido
```

Ilustración 6 Registro de usuarios

```python
def email_valido(s):
    return re.match(r"^[A-Za-z0-9._%+-]+@gmail\.com$", s) is not None
```

Ilustración 7 Función email_valido():

Validación de contraseñas

El sistema valida que la contraseña tenga al menos 8 caracteres y contenga letras mayúsculas, minúsculas y números.

```
=== POLIDELIVERY ===
1. Iniciar sesión
2. Registrarse
3. Salir
Opción: 2
Nombre: Alexis Chasi
Identificación: 0202659272
Edad: 19
Email (@gmail.com): aleix@gmail.com
Contraseña: hola1234
Contraseña débil
```

Ilustración 8 Validar contraseñas

```python
def password_valida(contrasenia):
    if len(contrasenia) < 8:
        return False
    return any("a" <= c <= "z" for c in contrasenia)
```

Ilustración 9 Función password_valida():

Autentificación de usuarios

Se verifican si el correo y la contraseña se encuentran en el archivo

```
=== POLIDELIVERY ===
1. Iniciar sesión
2. Registrarse
3. Salir
Opción: 1
Email: x
Contraseña: z
Credenciales incorrectas
```

Ilustración 10 Autenticación de usuarios

```python
def login(users):
    email = input("Email: ").strip()
    pw = input("Contraseña: ").strip()
    for u in users:
        if u["email"] == email and u["password"] == pw:
            return u
    return None
```

Ilustración 11 Función login():

Tabla 1 Validaciones y Seguridad

10. Conclusiones:

1. En conclusión, el programa “PoliDelivery” cumple con el objetivo de optimizar la gestión del proceso de entregas, permitiendo organizar pedidos, registrar información relevante y administrar rutas de manera estructurada.

2. Durante su desarrollo, se aplicaron conceptos esenciales de programación, como estructuras de datos, algoritmos de recorrido y técnicas de modernas que contribuyeron a la construcción de un sistema funcional.

3. Este proyecto constituye una base importante para futuras ediciones, ya que puede evolucionar dentro del ámbito de la logística y distribución.

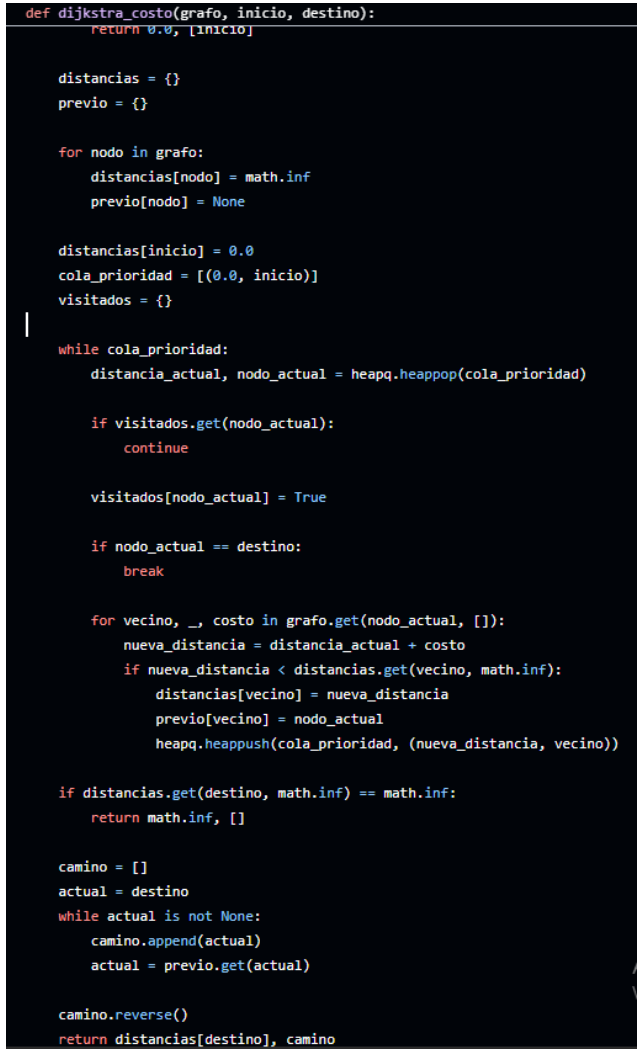
11. Recomendaciones:

1. Finalizado el desarrollado el sistema “PoliDelivery”, es recomendable reforzar el control de la entrada de datos mediante validaciones más exigentes, con el propósito de reducir errores en el registro de clientes, pedidos y rutas de entrega.

2. Asimismo, sería conveniente integrar una base de datos que permita almacenar toda la información de manera persistente, garantizando un manejo más seguro y eficiente de los registros a largo plazo.

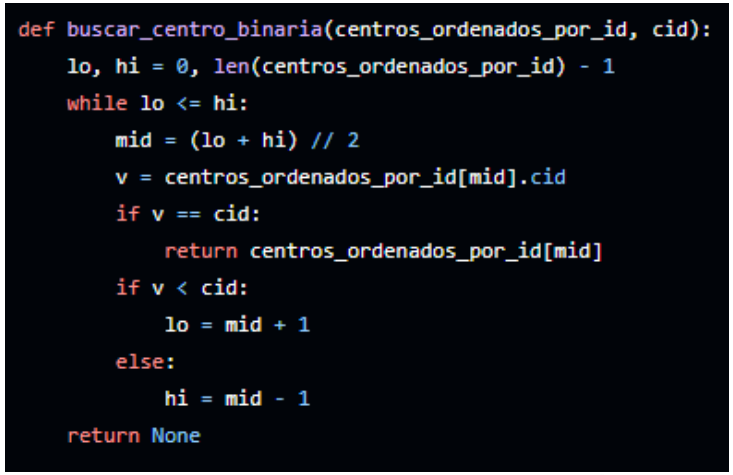
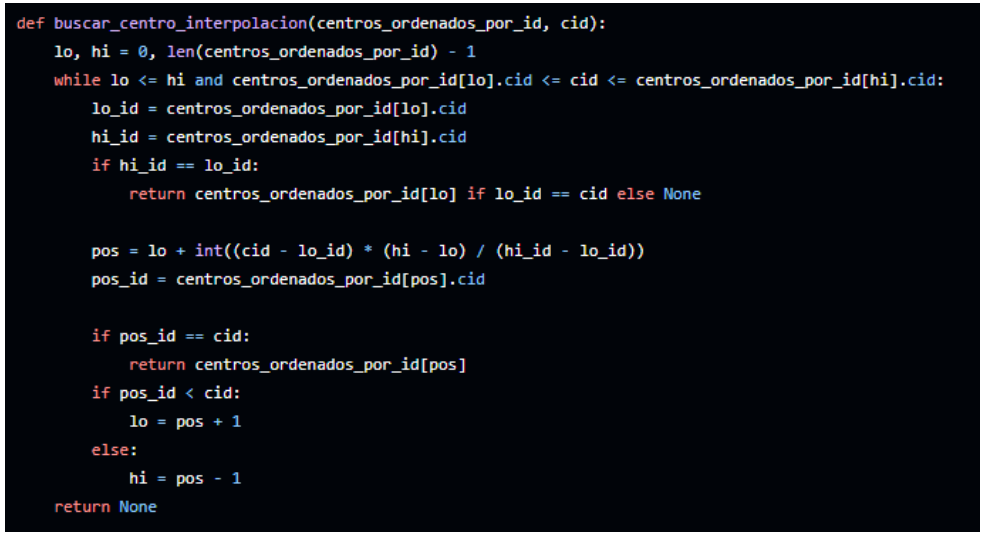
3. Finalmente, se sugiere implementar mejoras adicionales como la generación de reportes automáticos, el monitoreo del estado de las entregas y el diseño de una interfaz gráfica más intuitiva, lo cual incrementaría la funcionalidad y facilidad de uso del sistema.

12. Anexos:

Nombre de la función	Descripción	La función (Código esencial)
dijkstra_costo	Calcula la ruta con el menor costo económico entre dos puntos usando una cola de prioridad.	

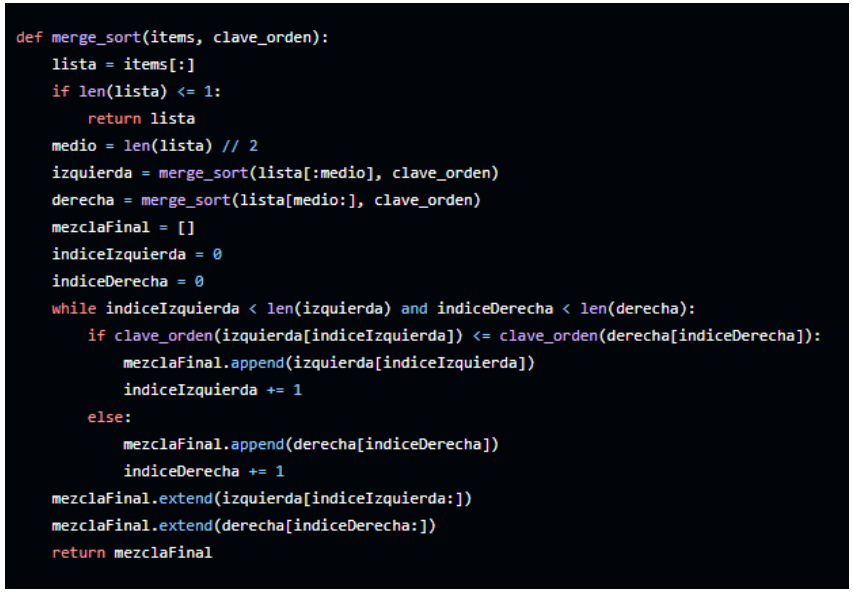
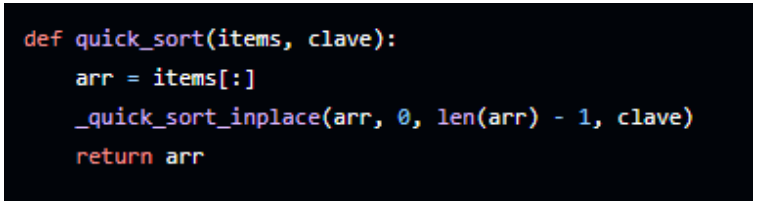
```python
def dijkstra_costo(grafo, inicio, destino):
    return 0.0, [inicio]

    distancias = {}
    previo = {}

    for nodo in grafo:
        distancias[nodo] = math.inf
        previo[nodo] = None

    distancias[inicio] = 0.0
    cola_prioridad = [(0.0, inicio)]
    visitados = {}

    while cola_prioridad:
        distancia_actual, nodo_actual = heapq.heappop(cola_prioridad)

        if visitados.get(nodo_actual):
            continue

        visitados[nodo_actual] = True

        if nodo_actual == destino:
            break

        for vecino, _, costo in grafo.get(nodo_actual, []):
            nueva_distancia = distancia_actual + costo
            if nueva_distancia < distancias.get(vecino, math.inf):
                distancias[vecino] = nueva_distancia
                previo[vecino] = nodo_actual
                heapq.heappush(cola_prioridad, (nueva_distancia, vecino))

    if distancias.get(destino, math.inf) == math.inf:
        return math.inf, []

    camino = []
    actual = destino
    while actual is not None:
        camino.append(actual)
        actual = previo.get(actual)

    camino.reverse()
    return distancias[destino], camino
```

Ilustración 12 Función dijkstra_costo():

Nombre de la función	Descripción	La función (Código esencial)
buscar_centro_binaria	Realiza una búsqueda rápida dividiendo la lista en mitades. Requiere que los datos estén ordenados.	 *Ilustración 13 Función buscar_centro_binaria():*
buscar_centro_interpolacion	Estima la posición del dato basándose en su valor numérico. Es más rápida que la binaria en datos uniformes.	 *Ilustración 14 Función buscar_centro_interpolacion():*

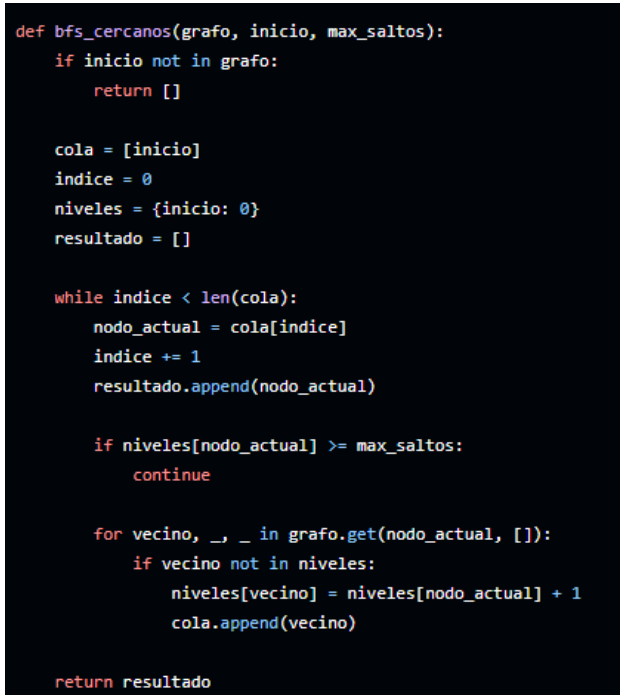
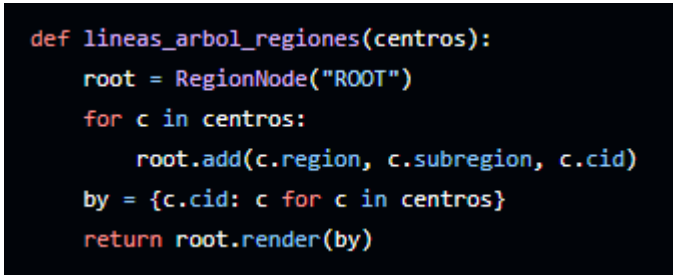
```python
def buscar_centro_binaria(centros_ordenados_por_id, cid):
    lo, hi = 0, len(centros_ordenados_por_id) - 1
    while lo <= hi:
        mid = (lo + hi) // 2
        v = centros_ordenados_por_id[mid].cid
        if v == cid:
            return centros_ordenados_por_id[mid]
        if v < cid:
            lo = mid + 1
        else:
            hi = mid - 1
    return None
```

```python
def buscar_centro_interpolacion(centros_ordenados_por_id, cid):
    lo, hi = 0, len(centros_ordenados_por_id) - 1
    while lo <= hi and centros_ordenados_por_id[lo].cid <= cid <= centros_ordenados_por_id[hi].cid:
        lo_id = centros_ordenados_por_id[lo].cid
        hi_id = centros_ordenados_por_id[hi].cid
        if hi_id == lo_id:
            return centros_ordenados_por_id[lo] if lo_id == cid else None

        pos = lo + int((cid - lo_id) * (hi - lo) / (hi_id - lo_id))
        pos_id = centros_ordenados_por_id[pos].cid

        if pos_id == cid:
            return centros_ordenados_por_id[pos]
        if pos_id < cid:
            lo = pos + 1
        else:
            hi = pos - 1
    return None
```

Nombre de la función	Descripción	La función (Código esencial)
merge_sort	Ordena los centros o rutas dividiendo la lista y mezclando las partes de forma recursiva.	 *Ilustración 15 Función merge_sort():*
quick_sort	Ordena los datos eligiendo un "pivote" y moviendo los elementos menores a un lado y mayores al otro.	 *Ilustración 16 Función quick_sort():*

```
def merge_sort(items, clave_orden):
    lista = items[:]
    if len(lista) <= 1:
        return lista
    medio = len(lista) // 2
    izquierda = merge_sort(lista[:medio], clave_orden)
    derecha = merge_sort(lista[medio:], clave_orden)
    mezclaFinal = []
    indiceIzquierda = 0
    indiceDerecha = 0
    while indiceIzquierda < len(izquierda) and indiceDerecha < len(derecha):
        if clave_orden(izquierda[indiceIzquierda]) <= clave_orden(derecha[indiceDerecha]):
            mezclaFinal.append(izquierda[indiceIzquierda])
            indiceIzquierda += 1
        else:
            mezclaFinal.append(derecha[indiceDerecha])
            indiceDerecha += 1
    mezclaFinal.extend(izquierda[indiceIzquierda:])
    mezclaFinal.extend(derecha[indiceDerecha:])
    return mezclaFinal
```

```
def quick_sort(items, clave):
    arr = items[:]
    _quick_sort_inplace(arr, 0, len(arr) - 1, clave)
    return arr
```

Nombre de la función	Descripción	La función (Código esencial)
bfs_cercanos	Explora los centros vecinos nivel por nivel para encontrar los más cercanos según el número de "saltos".	
lineas_arbol_regiones	Organiza y muestra los centros de forma jerárquica: Región > Subregión > Centro.	

```python
def bfs_cercanos(grafo, inicio, max_saltos):
    if inicio not in grafo:
        return []

    cola = [inicio]
    indice = 0
    niveles = {inicio: 0}
    resultado = []

    while indice < len(cola):
        nodo_actual = cola[indice]
        indice += 1
        resultado.append(nodo_actual)

        if niveles[nodo_actual] >= max_saltos:
            continue

        for vecino, _, _ in grafo.get(nodo_actual, []):
            if vecino not in niveles:
                niveles[vecino] = niveles[nodo_actual] + 1
                cola.append(vecino)

    return resultado
```

Ilustración 17 Función bfs_cercanos():

```python
def lineas_arbol_regiones(centros):
    root = RegionNode("ROOT")
    for c in centros:
        root.add(c.region, c.subregion, c.cid)
    by = {c.cid: c for c in centros}
    return root.render(by)
```

Ilustración 18 Función lineas_arbol_regiones():

Tabla 2 Algoritmos Implementados

Enlace Video: https://drive.google.com/file/d/1Sl_ghW5vEfaLR4R6bwlh8iNLDul1ohQ0/view?usp=sharing

Enlace GitHub: https://github.com/Alexis-Ch15/Proyecto-Algoritmos.git

Enlace Diapositivas:
https://www.canva.com/design/DAHAVH_k6ME/fMbNvmd9mjoqspjmaWThdg/edit?utm_content=DAHAVH_k6ME&utm_campaign=designshare&utm_medium=link2&utm_source=sharebutton