Ingeniería Eléctrica
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS UNIVERSIDAD DE CHILE

MyContactHub: Agenda electrónica

PROYECTO FINAL

UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE INGENIERÍA ELÉCTRICA
EL4203 - PROGRAMACIÓN AVANZADA

Profesor:	Pablo Martín.
Auxiliar:	Esteban Muñoz M.
Ayudantes:	Agustín González
	Diego Torreblanca
	Ignacio Romero Aravena
Integrante:	Matías Carvajal

Fecha: 13 de diciembre de 2024
Santiago de Chile

1. Descripción del problema abordado

En la actualidad, más del 65 % de las personas posee un teléfono celular con conexión a internet. Considerando la cantidad de relaciones que se establecen en el trabajo, los estudios o el ámbito personal, es razonable esperar que el número de contactos en la agenda telefónica de una persona sea considerable. De esta realidad surge la necesidad de una gestión eficiente de contactos, que permita agregarlos, eliminarlos y buscarlos con facilidad y en un tiempo razonable.

Las soluciones físicas, como agendas o cuadernos, no son suficientes para manejar la gran cantidad de información que se requiere almacenar. Por otro lado, las agendas electrónicas suelen formar parte de otros sistemas, lo que no garantiza un funcionamiento óptimo. Por esta razón, *MyContactHub* busca ofrecer una solución efectiva a estos problemas mediante una gestión eficiente de los datos y una interfaz fácil de usar.

1.1. Estado del Arte

En 1984 llegó al mercado lo que se considera la primera **PDA** (o agenda electrónica de bolsillo) del mundo: la *Organiser* de *Psion* (ver Figura 1.a). Sus especificaciones eran modestas y su diseño recordaba al de una calculadora, pero destacaba por su facilidad de uso y la posibilidad de desarrollar aplicaciones personalizadas [2]. Este *gadget* fue ganando popularidad con el tiempo, marcando el inicio de una nueva era tecnológica.

(a) Psion Organiser I de 1984 [1].

(b) Apple Newton MessagePad de 1993 [1].

A finales de los años 90 y principios de los 2000, surgieron tecnologías innovadoras, como las pantallas táctiles, el reconocimiento de escritura manual y la reproducción multimedia, lo que hizo a las PDA aún más robustas y funcionales, como se ve en la Figura 1.b. Sin embargo, la aparición de los *smartphones* marcó el declive definitivo de las PDA, ya que *"todo lo que ofrecían se podía disfrutar con un smartphone"*, según cuenta Ros en el sitio web *MuyComputer* [2].

1.2. Objetivo

Se busca diseñar la agenda electrónica *MyContactHub* para satisfacer las necesidades de un caso específico: organizar y gestionar eficientemente un gran número de contactos. El objetivo principal es centralizar esta información de forma flexible y accesible, particularmente en contextos donde el volumen de contactos es elevado, como en el ámbito laboral o académico.

En la actualidad, el amplio espectro de medios de comunicación disponibles ha fragmentado la manera en que las personas almacenan y acceden a los datos de sus contactos. Por ello, la utilidad principal de *MyContactHub* radica en su capacidad para consolidar toda esta información en un solo lugar. La agenda permitirá integrar datos provenientes de diferentes plataformas, como correos electrónicos, números telefónicos y redes sociales, brindando a los usuarios una solución centralizada que ahorra tiempo y mejora la organización.

2. Propuesta

2.1. ¿Qué es lo que se hará?

Se propone desarrollar una agenda electrónica, denominada *MyContactHub*, que permita reunir y gestionar eficientemente la información de distintos contactos en una aplicación sencilla, con operaciones eficientes, que optimice la experiencia del usuario.

La propuesta de interfaz gráfica se muestra en la Figura 2. Como se aprecia, el diseño prioriza la simplicidad y la usabilidad, presentando un conjunto de opciones claramente organizadas. Cada funcionalidad incluida en la agenda cuenta con una justificación técnica y práctica, garantizando que la aplicación responda a las necesidades específicas del caso planteado.

Figura 2: Primera versión de la interfaz gráfica de *MyContactHub*, en donde se incluye un menú principal (izquierda) y dos ventanas emergentes (derecha) que permiten diversos ajustes relacionados a un contacto.

- **Agregación, búsqueda y borrado:** operaciones básicas que debería hacer cualquier agenda que busque organizar contactos.
- **Búsqueda personalizada:** dado que un contacto tiene más de un atributo y no necesariamente se saben todos de ellos, es vital tener distintas formas de buscar a la persona objetivo.
- **Ordenar por:** similar a lo anterior, un ordenamiento (dependiente de cierto atributo) permite una correcta visualización de los datos para mejorar la búsqueda.
- **Agregar atributo:** es posible que se quieran agregar notas, etiquetas o redes sociales no convencionales. Por ende, una lista de características propuesta por el mismo usuario permite una mejor personalización y utilidad dentro de la agenda.
- **Modificar:** los datos de las personas varían en el tiempo. Es indispensable tener una forma de modificar los datos para mantener actualizado el contacto.
- **Otros:** además de lo anterior, se incluyen botones para marcar contactos, desplazarse entre páginas y se explicita la versión actual que se está usando.

2.2. ¿Cómo se hará?

Siguiendo los principios enseñados en el curso, la implementación se realizará en `Python`, basándose en el diagrama de flujo presentado en la Figura 3. Las funcionalidades principales, descritas previamente, se implementarán en esta etapa, y se detallarán a continuación.

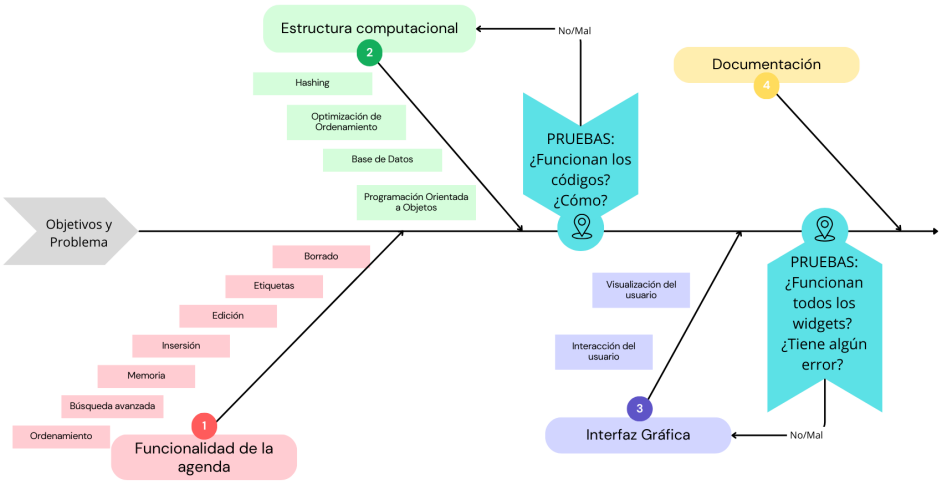

Figura 3: Diagrama de flujo del trabajo. El proyecto va avanzando a través de la linea central, a medida que se van completando las etapas. Existen *checkpoints* en celeste, que indican una revisión exhaustiva antes de avanzar.

2.2.1. Estructura computacional

- **Hashing:** a medida que aumenta el número de contactos, es crucial implementar una estructura eficiente para buscar, insertar y eliminar elementos. Las tablas *hash* son ideales para estas tareas debido a su velocidad en operaciones clave. Existen múltiples formas de implementación, como el *doble hashing*, que minimiza colisiones, o el encadenamiento, que conserva la posición de elementos.

  Inicialmente, se utilizará un diccionario nativo de `Python`, que implementa *hashing* en sus operaciones básicas y ofrece un rendimiento adecuado [3]. En caso de que esta solución no sea suficiente, se explorarán opciones más avanzadas, como tablas *hash* personalizadas.

- **Optimización de Ordenamiento:** para garantizar una ordenación eficiente de los contactos, se evaluarán algoritmos con complejidad $O(n \log n)$, como *QuickSort* [4]. Sin embargo, para texto se utilizará la función `sorted` de `Python`, que implementa *TimSort* y ofrece un rendimiento superior para ordenar cadenas [5]. De todas formas, se debe probar en que circunstancias es un algoritmo mejor que otro.

- **Base de Datos:** la memoria persistente se manejará inicialmente mediante archivos `.csv`, `.pkl` o `.json`, los cuales son portátiles y simples de gestionar. La elección final dependerá de las necesidades específicas del proyecto, como flexibilidad y jerarquía de datos.

- **Programación Orientada a Objetos:** este paradigma permitirá estructurar y almacenar de manera eficiente las características de cada contacto. Los contactos compartirán una estructura común mediante clases, facilitando su gestión.

2.2.2. Interfaz y memoria

Una vez que el código haya sido probado en diversas condiciones normales de funcionamiento, la interfaz será diseñada utilizando la librería `Tkinter` [6]. Esta librería permitirá implementar una interfaz gráfica de usuario (GUI) que facilite la interacción intuitiva con cada una de las funciones desarrolladas, garantizando resultados eficientes y una experiencia de usuario amigable. Es fundamental aprovechar la programación orientada a objetos (POO) para estructurar el proyecto de tal forma que sea fácil de mantener y modificar.

Por otro lado, las acciones realizadas por el usuario deben ser almacenadas en memoria de manera persistente. Una opción viable sería el uso de una base de datos basada en archivos `.csv`, que destaca por su simplicidad y portabilidad. Sin embargo, el modelo relacional que ofrece un archivo `.csv` puede ser demasiado rígido para aplicaciones más complejas, lo que podría limitar la flexibilidad del diseño.

Como alternativa, se propone el uso de archivos `.json`, que son más flexibles y permiten jerarquia. Este formato es especialmente útil si los contactos requieren atributos adicionales o relaciones más complejas, como múltiples números. Además, Python ofrece soporte nativo para trabajar tanto con `csv` como con `json` o **pkl** mediante las librerías estándar `csv`, `json` y `pickle`, lo que simplifica la implementación y reduce la necesidad de dependencias externas.

2.2.3. Documentación

Con el programa realizado, se vuelve posible (y recomendado) hacer una documentación, aunque breve, del programa. Sumado a esto, gracias a `auto-py-to-exe` [7] es posible convertir el programa en un archivo binario o ejecutable para Windows. De manera ideal, un repositorio en *GitHub* permitiría tener el programa actualizado y utilizable para otras personas.

2.3. Problemas encontrados

Existe una gran variedad de problemas que se podrían generar durante la implementación del programa. Por ejemplo, se deben verificar distintos algoritmos para poder encontrar aquel que sea más eficiente. Por otro lado, `Tkinter` a pesar de ser sencillo, debe ser bien integrado con el código. Por último, el manejo de las base de datos es complejo y podría generarse corrupción de archivos, colisiones, perdidas de datos, etcetera. Sumado a que su seguridad es practicamente nula.

3. Implementación

Tal como se mencionó en secciones anteriores, los *scripts* y programas fueron documentados en un repositorio público en *GitHub*, disponible en el siguiente enlace. En particular, se utilizaron dos archivos de código principales: `classes` e `interface`, que contienen, respectivamente, las clases relacionadas con la agenda y la interfaz gráfica.

3.1. Clase Contact

Esta clase representa cada contacto en la agenda. Almacena los atributos del contacto en un diccionario denominado *atributes* y define su identidad en la variable *identity*, que consiste en su nombre completo, incluidos los apellidos. Para inicializar un objeto de esta clase, se requieren los siguientes datos personales:

- `name`, los nombres de la persona;
- `last_name`, sus apellidos;
- `mail`, su dirección de *email*;
- y `number`, su número teléfonico.

Por otro lado, la clase incluye dos métodos principales: `show()`, que permite visualizar los atributos registrados de la persona, y `add(key, value)`, que facilita la incorporación de un nuevo atributo `key` con su correspondiente valor `value`. La implementación de estos métodos se presenta en el Código 1, mostrado a continuación.

Código 1: Clase `Contact`, almacena un contacto y sus atributos. cod:contacts

```
1  class Contact:
2      def __init__(self,
3                   name: str,
4                   last_name: str,
5                   mail: str,
6                   number: str):
7          self.atributes = {'Nombres': name,
8                            'Apellidos': last_name,
9                            'Email': mail,
10                           'Teléfono': number}
11         self.identity = name + ' ' + last_name
12
13     def show(self):
14         for key in self.atributes:
15             print(key + ':', self.atributes[key])
16
17     def add(self, key: str, value):
18         self.atributes[key] = value
```

3.2. Clase Agenda

Esta clase representa la agenda, la cual contiene un diccionario `contacts` que almacena los contactos y una variable `num_contacts` que registra la cantidad total de contactos. A continuación, se describen los métodos disponibles junto con sus respectivos códigos:

- `add`: Este método recibe un contacto (de tipo `Contact`) y lo incorpora a la agenda, utilizando su `identity` como llave. Devuelve `True` si la operación se realiza con éxito, o `False` si el contacto ya existe en la agenda, en cuyo caso no se realiza la adición.

Código 2: Método `add` para agregar contactos a la agenda.

```
    def add(self, contact: Contact):
        if self.contacts.get(contact.identity) is None:
            self.contacts[contact.identity] = contact
        else:
            print('El contacto ' + contact.identity + ' ya está en la agenda')
            return False
        print('Agregado ' + contact.identity)
        self.num_contacts += 1
        return True
```

- `remove`: Este método recibe una `identity` y elimina el contacto correspondiente. Devuelve `False` si el contacto no se encuentra en la agenda, y `True` si la eliminación se realiza con éxito.

Código 3: Método `remove` para eliminar un contacto de la agenda.

```
    def remove(self, key: str):
        try:
            contact = self.contacts.pop(key)
        except KeyError:
            print('El contacto no está en la agenda')
            return False
        print('Eliminado ' + contact.identity)
        self.num_contacts -= 1
        return True

```

- `show`: Este método recibe una `identity` y, en caso de existir, muestra el contacto junto con sus atributos. Devuelve el contacto si se encuentra en la agenda; en caso contrario, retorna `None`. Aunque este método no tiene un impacto significativo en el programa, es útil para realizar *debugging* y observar resultados dentro de `Python`.

Código 4: Método `show`, que hace un `print` del contacto.

```
    def show(self, key: str):
        contact = self.contacts.get(key)
        if contact is None:
            print('El contacto no está en la agenda')
        else:
            contact.show()
        return contact
```

- `search`: Este método recibe un `string`, que puede ser una `identity` o el valor de un `atribute`, junto con el `atribute` a buscar, si es necesario. Por defecto, realiza la búsqueda por `identity`

y `atribute` está configurado como `None`. Retorna una lista con las `identity` de los contactos encontrados. Este método puede clasificarse como una **búsqueda lineal**, destacando las siguientes ventajas:

1. **Normalización**: No distingue entre mayúsculas y minúsculas.
2. **Coincidencias parciales**: Busca coincidencias dentro de los primeros caracteres, sin exigir que coincida todo el atributo.
3. **Condicionalidad**: Permite centrar la búsqueda en un atributo específico, en lugar de limitarse únicamente a las llaves.

Por último, cabe destacar que si solo se encuentra un elemento, el método retorna directamente su `identity`, en lugar de una lista.

Código 5: Método `search`, que busca un contacto en función de alguna de sus características.

```
    def search(self, key_search: str, atribute=None):
        n = len(key_search)
        if atribute is not None:
            keys_finds = [key for key, contact in self.contacts.items() if len(contact.atributes.
↪ get(atribute)) >= n
                          and contact.atributes.get(atribute)[: n].lower() == key_search.lower()]
            return keys_finds if len(keys_finds) != 1 else keys_finds[0]
        keys_finds = []
        for key in self.contacts:
            if len(key) >= n and key[: n].lower() == key_search.lower():
                keys_finds.append(key)
                print(key)
        return keys_finds if len(keys_finds) != 1 else keys_finds[0]
```

- `sort`: Este método recibe un `atribute` y ordena los contactos en función de dicho atributo. Si no se especifica un `atribute` (es decir, si es `None`), los contactos se ordenan según su `identity`. Este método no realiza retornos. Una de sus principales ventajas es la capacidad de ordenar por cualquier atributo, incluso por aquellos que no están presentes en todos los contactos.

Código 6: Método `sort`, que ordena los contactos en función de algún atributo.

```
    def sort(self, atribute=None):
        if atribute is None:
            self.contacts = dict(sorted(self.contacts.items()))
            return
        self.contacts = dict(sorted(self.contacts.items(), key=lambda item: item[1].atributes.
↪ get(atribute, chr(0x10FFFF))))

```

- `show_all`: Este método muestra las `identity` de todos los contactos almacenados en la agenda. No recibe parámetros ni realiza retornos. Similar al método `show`, tiene un impacto menor en el programa, pero es útil para visualizar la lista completa de contactos.

Código 7: Método `show_all` para mostrar todos los contactos.

```
    def show_all(self):
        for key in self.contacts:
            print(key)

```

- `copy`: Este método crea una copia completa de la agenda, incluyendo todos los contactos y el número total de estos. Retorna la copia generada.

Código 8: Método `copy` para hacer una copia de la agenda.

```
    def copy(self):
        copy = Agenda()
        copy.contacts = self.contacts.copy()
        copy.num_contacts = self.num_contacts
        return copy

```

3.3. Rendimiento

Para evaluar la eficiencia de cada método, se utilizó la librería `Faker` [8] para generar contactos aleatorios. Se realizaron diversas pruebas que abarcaron las principales funciones implementadas. Los resultados obtenidos se presentan en la Tabla 1, donde se evidencia el reducido tiempo requerido para ejecutar cada método. Cabe destacar que el mayor tiempo registrado fue de 3.6966 ± 0.3319, correspondiente a la operación de inserción.

Por otro lado, la precisión en la cantidad de contactos generados confirma que la elección de la `identity` como clave fue adecuada, sin considerar otros factores como la prevalencia de ciertos apellidos o las variaciones en la escritura.

Tabla 1: Resultados obtenidos a través de pruebas de rendimiento con 10 repeticiones en cada una. Se presentan con el formato promedio ± desviación estándar.

Prueba	**Cantidad de contactos**	**Tiempo [s]**
Búsqueda por identidad	1000 ± 0	1.401 ± 0.8032
Búsqueda por atributo	1000 ± 0	0.6051 ± 0.1784
Eliminación	999.8 ± 0.4	0.091 ± 0.0591
Inserción	1000 ± 0	3.6966 ± 0.3319
Ordenación por identidad	999.7 ± 0.4	0.0013 ± 0.0005
Ordenación por atributo	999.7 ± 0.4	0.002 ± 0.0008

3.4. Memoria

Entre las distintas opciones propuestas inicialmente, se decidió aprovechar las funcionalidades de `Python` para guardar los datos en un archivo `pickle`. Esta elección se basó en varios factores:

- **Comodidad:** No se requieren cambios en la estructura, y los métodos para guardar y cargar los datos son directos y sencillos de implementar.
- **Tiempos:** Las pruebas realizadas con una agenda de 10000 contactos arrojaron un tiempo de guardado de 0.0349 [s] y un tiempo de carga de 0.108 [s], lo cual resulta imperceptible para el usuario.
- **Tamaño:** En las mismas pruebas, el archivo generado ocupó únicamente 1.163531 [MB], un tamaño considerablemente reducido en relación con la cantidad de contactos almacenados.

3.5. Interfaz

La interfaz gráfica fue implementada mediante la clase `Interface`, utilizando su memoria interna a través del uso de `self`. Debido a la gran cantidad de utilidades y *widgets* necesarios, se describirán únicamente de manera general su inicialización y funcionalidades principales. La interfaz se muestra en la Figura 4.

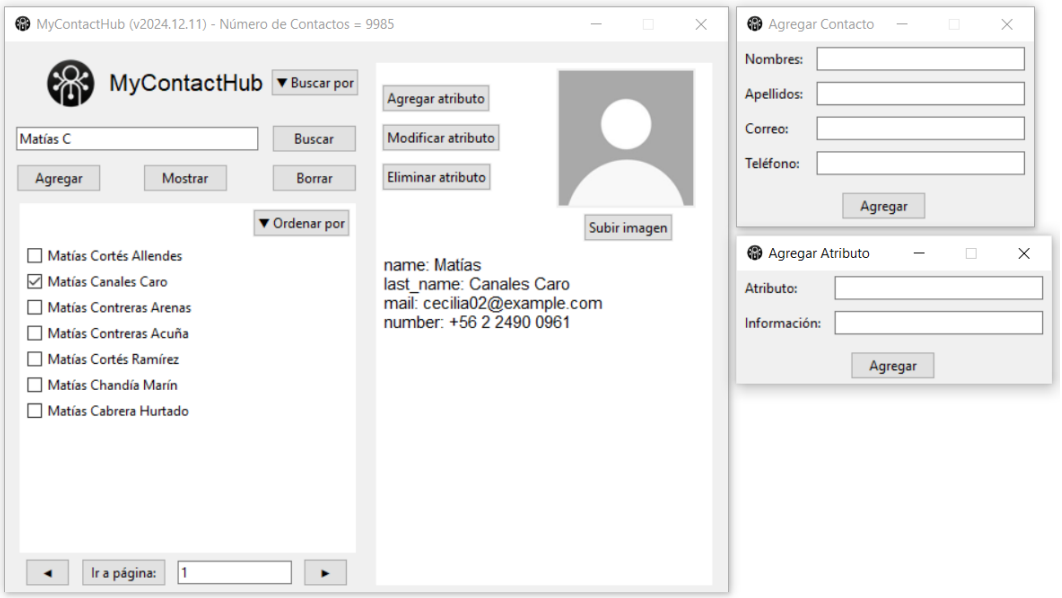

Figura 4: Interfaz Gráfica del programa *MyContactHub*. A la derecha, se muestran parte de sus ventanas emergentes (no todas). Los datos personales o que involucren otras personas fueron censurados.

Por otro lado, dentro de las funcionalidades que ofrece la ventana mostrada en la Figura 4 se incluyen:

- **Búsqueda:** Es posible realizar búsquedas por atributo o por identidad, dependiendo de los requerimientos del usuario. Los resultados se mostrarán en un cuadro que lista los contactos correspondientes.
- **Visualización:** Para gestionar la visualización, se almacenan las llaves de los contactos, su *widget* correspondiente y una variable *booleana*. Esta última indica si el contacto ha sido seleccionado por el usuario, permitiendo realizar acciones específicas sobre él. La visualización se gestiona mediante botones de *check* junto al nombre completo (identidad) del contacto.

- **Indicadores:** Se dispone de indicadores para la búsqueda, la ordenación y el contacto actualmente mostrado. Estos indicadores permiten registrar y visualizar el estado del programa en cada momento.
- **Respaldos:** Los contactos encontrados u ordenados se manejan de manera separada para evitar sobrescribir la agenda principal, preservando tanto su estructura original como su orden temporal.
- **Organización:** Los datos están organizados por páginas, lo que facilita su navegación y permite desplazarse de manera ordenada o según el orden de inserción.
- **General:** Tal como se mencionó anteriormente, el programa hereda todas las funcionalidades adicionales de las clases con las que fue construido, incluyendo la inserción, eliminación y modificación de atributos. Esto hace que la agenda sea altamente personalizable. Además, admite la subida de imágenes que permitan identificar mejor a los contactos.

Además, en el extracto del Código 9 se presentan las variables dinámicas que posibilitan la ejecución de las acciones descritas anteriormente. Tanto la agenda (clase) como las imágenes (diccionario) se almacenan directamente en un archivo `pickle`, como se explicó en la sección anterior.

Código 9: Extracto del código de inicialización para la interfaz.

```
class Interface:
    def __init__(self):
        '''
        agenda y otras variables
        '''
        self.agenda = Agenda()  # Agenda
        self.atr_busqueda = None  # Atributo de busqueda, por defecto 'None' busca identidad
        self.contacts_cb = []  # CheckButtons que se muestran como contactos
        self.contacts_bv = []  # BooleanVars de los CheckButtons que se muestran
        self.contacts_id = []  # Identidad de cada contacto mostrado
        self.busqueda_value = False  # ¿'Se hizo busqueda?
        self.busqueda = []  # Valores buscados
        self.orden_value = False  # ¿'Se ordenó?
        self.ordenados = []  # Valores ordenados
        self.actually_key = []  # Key del contacto actual que se muestra
        self.images = {}
```

Para finalizar, el código completo se encuentra disponible en el repositorio mencionado anteriormente. En este repositorio, se incluyen las instrucciones para el uso del programa y el ejecutable para *Windows*, generado con `auto-py-to-exe`. Con la implementación actual, los pasos futuros consistirán en utilizar la agenda para identificar posibles errores en la implementación y proponer mejoras.

Referencias

[1] Wikipedia, PDA. Fundación Wikimedia, 2024, https://es.wikipedia.org/wiki/PDA.

[2] Ros, I., Un trocito de historia: evolución y muerte de las PDA. MuyComputer, 2014, https://www.muycomputer.com/2014/07/06/un-trocito-de-historia-evolucion-y-muerte-de-las-pda/.

[3] Python Software Foundation, The Python Standard Library. Built-in Types. Python 3.10 Documentation, 2024, https://docs.python.org/3/library/stdtypes.html#dict.

[4] Sipiran, I., Algoritmos y Estructuras de Datos. Departamento de Ciencias de la Computación. Universidad de Chile, 2024, https://github.com/ivansipiran/AED-Apuntes/.

[5] Python Software Foundation, The Python Standard Library. Built-in Functions. Python 3.10 Documentation, 2024, https://docs.python.org/3/library/functions.html#sorted.

[6] Python Software Foundation, Graphical User Interfaces with Tk. Python 3.13 Documentation, 2024, https://docs.python.org/3/library/tk.html.

[7] Geeks for Geeks, Introduction to auto-py-to-exe Module. Web Site, 2024, https://www.geeksforgeeks.org/introduction-to-auto-py-to-exe-module/.

[8] Faraglia, D., Faker 33.1.0 Documentation. Sitio web, 2014, https://faker.readthedocs.io/en/master/.