

Edson Leonel Flores García

Octubre 2025

Este manual técnico detalla la arquitectura, diseño, componentes clave y consideraciones técnicas del "Gestor de Notas Simple", una aplicación diseñada para la organización y recuperación eficiente de notas.

El sistema provee una herramienta ágil para la gestión de notas, incluyendo categorización, búsqueda avanzada y sincronización básica. Este documento explora las decisiones de ingeniería detrás de cada módulo y sus implicaciones en rendimiento y escalabilidad, ofreciendo una comprensión profunda para desarrolladores y administradores de sistemas.

Visión General Técnica del Proyecto

El Gestor de Notas Simple se desarrolló con un enfoque modular para facilitar la mantenibilidad y expansión. Utiliza una pila tecnológica moderna para un rendimiento óptimo. Incluye un subsistema de autenticación/autorización, un gestor de bases de datos para persistencia, una API RESTful para el frontend, y una interfaz de usuario (UI) responsiva.

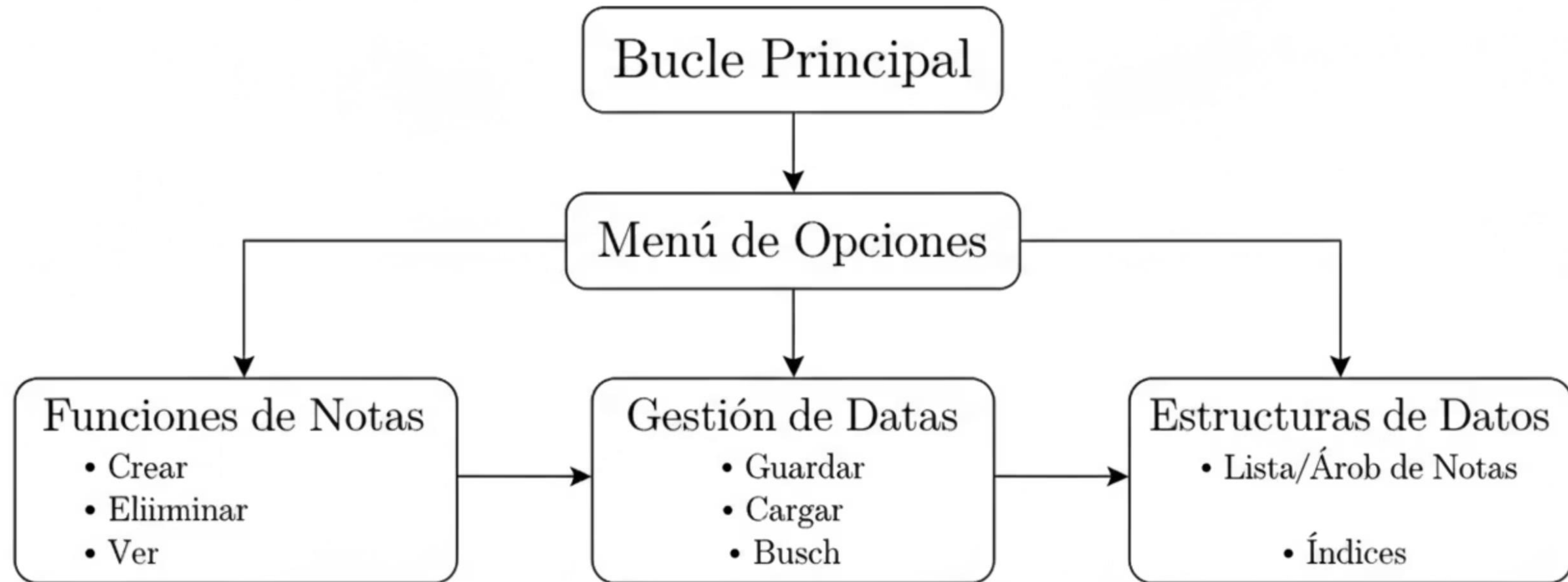
El backend implementa un patrón Modelo-Vista-Controlador (MVC) para la separación de preocupaciones. El frontend emplea un framework reactivo para actualizaciones dinámicas de la interfaz, optimizando la experiencia de usuario.

Componentes Clave de la Arquitectura

- Módulo de Autenticación:** Verifica la identidad del usuario y gestiona permisos mediante autenticación basada en tokens (JWT), asegurando sesiones seguras y escalables.
- Servicio de Notas:** Gestiona las operaciones CRUD (Crear, Leer, Actualizar, Borrar) de las notas, incluyendo lógica para categorización, etiquetado y búsqueda de texto completo.
- Base de Datos:** PostgreSQL, una base de datos relacional robusta, garantiza la integridad de los datos, permite consultas complejas y soporta JSONB para metadatos flexibles.
- API RESTful:** Sirve como punto de entrada principal. Define endpoints claros y semánticos, utilizando JSON para el intercambio de datos.
- Interfaz de Usuario:** Desarrollada con React.js, ofrece una experiencia intuitiva y adaptable a diversos dispositivos (escritorio y móvil) mediante un diseño responsivo.

Cada componente prioriza la resiliencia, incorporando mecanismos de manejo de errores y logging para una depuración y monitoreo eficientes. La comunicación entre módulos utiliza patrones asíncronos cuando es posible para mejorar la capacidad de respuesta general del sistema.

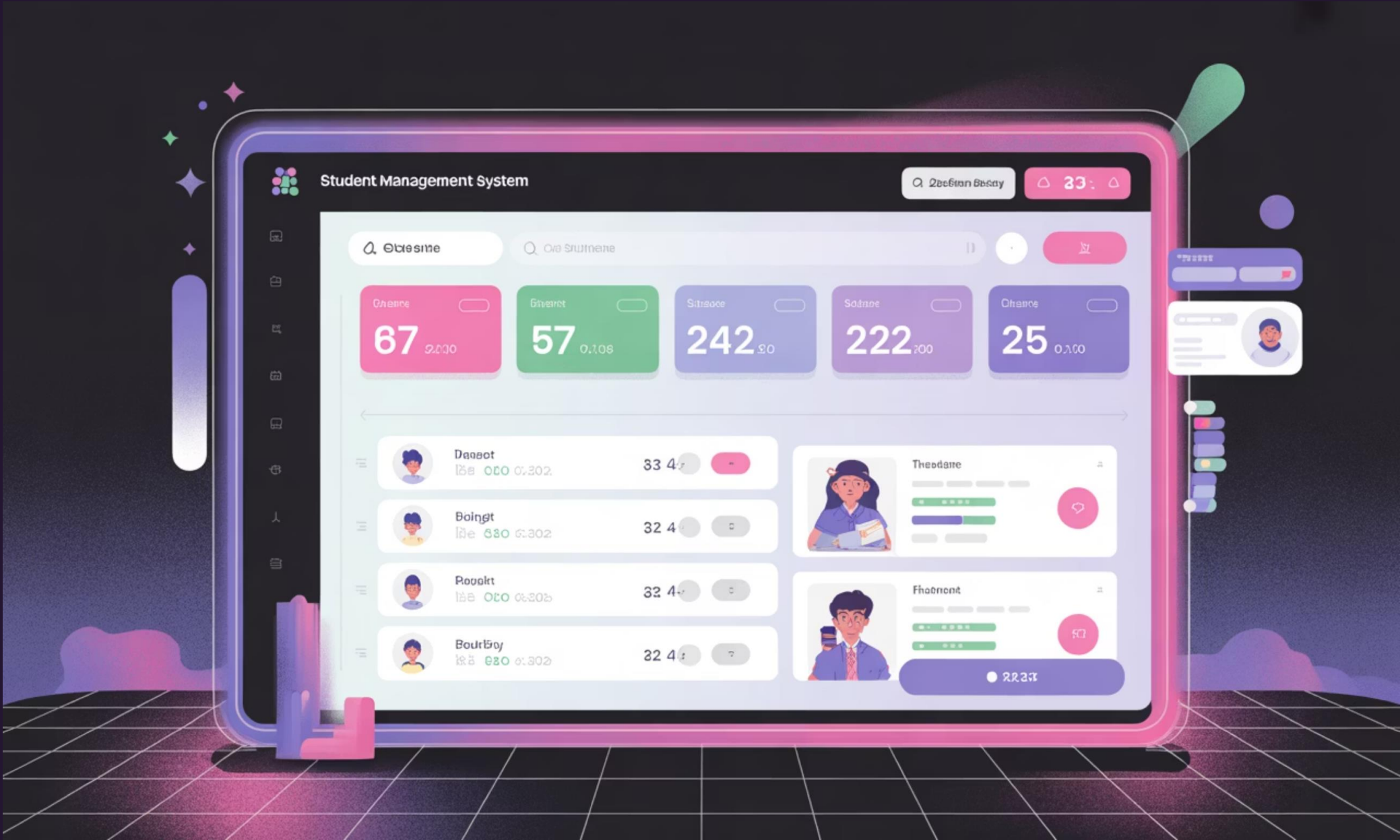
DIAGRAMA DEL SISTEMA GESTOR DE NOTAS



Descripción Técnica General del Sistema

El **Gestor de Notas Simple** es una aplicación de consola desarrollada en Python para la gestión académica personal. Facilita el control de asignaturas y calificaciones mediante un diseño que prioriza la eficiencia y la claridad en la presentación de información.

La implementación utiliza **estructuras de datos** como listas dinámicas (para asignaturas) y colas (para eventos/historial). Emplea **algoritmos** de búsqueda (lineal y binaria) y ordenamiento (burbuja, selección, inserción, o Quicksort/Mergesort) para optimizar la localización y organización de datos, mejorando la eficiencia del sistema. Incluye funcionalidades avanzadas como simulación de solicitudes y un sistema de historial de cambios para trazabilidad.



Estructura General del Código

La arquitectura del **Gestor de Notas Simple** se ha diseñado modularmente, promoviendo la separación de responsabilidades y la encapsulación. Esta organización facilita la comprensión, el mantenimiento y futuras expansiones, asegurando que cada componente cumpla una función específica y bien definida.

01

Definición de Estructuras de Datos Fundamentales

El sistema inicializa estructuras de datos clave para la gestión de la información en memoria:

- `mis_asignaturas``: Lista principal que almacena `Asignatura` objetos.
- `historial``: Cola (FIFO) para registrar cambios y operaciones importantes, actuando como un log de auditoría.
- `solicitudes``: Cola (FIFO) para simular el procesamiento secuencial de peticiones.

Esta configuración permite una manipulación directa y flexible de los datos principales.

03

Funciones de Gestión: Modularidad y Encapsulación

Cada funcionalidad se encapsula en una función independiente (ej. `agregar_asignatura()`, `calcular_promedio()`, `ordenar_asignaturas()`, `buscar_asignatura_lineal()`). Este enfoque modular mejora la legibilidad, facilita la depuración y promueve la reutilización del código. Permite el desarrollo y prueba independiente de cada característica y simplifica la incorporación de nuevas funcionalidades sin afectar el resto del sistema.

02

Menú Principal: Interfaz de Usuario en Consola

La función `menu()` es el componente central para la interacción con el usuario. Presenta todas las opciones disponibles (agregar, listar, modificar, eliminar asignaturas, calcular promedios) y captura la entrada del usuario para dirigir el flujo del programa. Su diseño busca la claridad para guiar al estudiante a través de las funcionalidades.

04

Bucle Principal de Ejecución: El Corazón de la Interacción

El sistema opera mediante un ciclo `while` continuo, el cual es el bucle principal de la aplicación. Este bucle ejecuta repetidamente:




1. Llama a `menu()` para mostrar opciones.
2. Espera la entrada del usuario.
3. Invoca la función específica según la opción elegida.

El ciclo se mantiene activo hasta la salida explícita del usuario, garantizando una interacción fluida. Incluye manejo de errores para opciones inválidas.



Uso de Estructuras de Datos

La elección de estructuras de datos es fundamental para la funcionalidad y eficiencia del sistema. Hemos implementado listas, colas y pilas para optimizar la manipulación de información académica y operativa. Esta sección detalla su rol, consideraciones técnicas y ventajas.

		
<div><h3>Listas</h3><p><code>mis_asignaturas</code> y <code>mis_notas</code> son listas de Python que almacenan nombres de asignaturas y sus calificaciones. La correspondencia entre asignatura y nota se mantiene por índice, facilitando su recuperación y actualización.</p><p>Consideraciones técnicas: Las listas de Python son arrays dinámicos. El acceso por índice es $O(1)$. La inserción o eliminación en posiciones intermedias es $O(n)$. Para adiciones al final y accesos frecuentes, son eficientes.</p></div>	<div><h3>Cola (FIFO)</h3><p>La lista <code>solicitudes</code> funciona como una cola FIFO (First-In, First-Out) para gestionar solicitudes de cambio o revisión de notas. Las solicitudes se añaden al final y se procesan en orden de llegada, asegurando equidad.</p><p>Consideraciones técnicas: Para colas eficientes en Python, se recomienda <code>collections.deque</code>. Ofrece operaciones de adición y eliminación (<code>enqueue/dequeue</code>) en ambos extremos con complejidad $O(1)$, a diferencia de <code>list.pop(0)</code> que es $O(n)$.</p></div>	<div><h3>Pila (LIFO)</h3><p>La estructura <code>historial</code> opera como una pila LIFO (Last-In, First-Out), registrando modificaciones o eventos significativos en las notas. Esto facilita la implementación de funciones de "deshacer" y la revisión de la secuencia inversa de operaciones.</p><p>Consideraciones técnicas: Las listas de Python son eficientes como pilas. Las operaciones <code>append()</code> (push) y <code>pop()</code> (pop del último elemento) tienen una complejidad de tiempo amortizada de $O(1)$, ideal para gestión de historial.</p></div>

Ventajas Clave de las Listas

- **Acceso Directo por Índice:** Recuperación $O(1)$ de elementos por su posición.
- **Manipulación Sencilla:** Operaciones de añadir, actualizar y eliminar son directas.
- **Correspondencia Implícita:** Relación clara entre asignatura y nota por índices paralelos.
- **Flexibilidad:** Crecimiento y decrecimiento dinámico según las necesidades.

Ventajas Clave de Colas y Pilas

- **Orden Cronológico (Colas):** Procesamiento justo y transparente de solicitudes (FIFO).
- **Gestión Eficiente del Historial (Pilas):** Seguimiento lógico de cambios, facilitando "deshacer" y auditorías (LIFO).
- **Modelado de Procesos:** Representación fiel de flujos de trabajo reales.
- **Rendimiento Optimizado:** Operaciones intrínsecas (push/pop, enqueue/dequeue) son generalmente $O(1)$.

Justificación de Algoritmos de Ordenamiento

Para mejorar la usabilidad y eficiencia en la gestión de datos académicos, se implementa el ordenamiento por nota (`ordenar_por_nota`) y por nombre (`ordenar_por_nombre`) usando el algoritmo de **ordenamiento burbuja**. Esta elección se basa en las siguientes consideraciones técnicas:

1	2	3
<h3>Simplicidad y Mantenimiento</h3> <p>El algoritmo burbuja es simple, de fácil implementación y comprensión, lo que minimiza errores y facilita el mantenimiento. Su implementación directa reduce la curva de aprendizaje para desarrolladores y optimiza la depuración. Su complejidad espacial es $O(1)$.</p>	<h3>Adecuación para Datos Pequeños</h3> <p>El sistema maneja listas de asignaturas y notas por estudiante de tamaño "pequeño" (típicamente $n < 50$). Aunque su complejidad temporal es $O(n^2)$, para estos volúmenes de datos el rendimiento es suficiente y no percibible por el usuario final, evitando la sobreingeniería.</p>	<h3>Ordenamiento Paralelo y Consistencia</h3> <p>Las listas <code>mis_asignaturas</code> y <code>mis_notas</code> mantienen una correspondencia estricta por índice. El burbuja permite un "ordenamiento paralelo": al intercambiar un elemento en una lista, se realiza el mismo intercambio en la otra, preservando la relación asignatura-nota sin lógica adicional compleja. Esto garantiza la integridad referencial de los datos.</p>

❏ **Consideración Técnica:** A pesar de que algoritmos como QuickSort o MergeSort ofrecen una complejidad temporal promedio de $O(n \log n)$, la elección del ordenamiento burbuja es una decisión de diseño consciente para volúmenes de datos pequeños. Para n reducidos, la sobrecarga constante de algoritmos más complejos puede hacer que el burbuja sea competitivamente rápido. La justificación principal reside en la simplicidad del código, facilidad de depuración, menor riesgo de errores y agilidad en el mantenimiento, factores críticos en aplicaciones del mundo real con requisitos de datos moderados. Las funciones `ordenar_por_nota` y `ordenar_por_nombre` implementan este algoritmo directamente sobre las listas paralelas, asegurando que las validaciones de entrada, como tipos de datos y rangos de notas, se realicen previo a la llamada a la función para evitar errores en el proceso de ordenamiento.

En síntesis, la selección del ordenamiento burbuja optimiza la funcionalidad de ordenamiento, considerando el volumen de datos, la simplicidad de desarrollo y la facilidad de mantenimiento. Su capacidad para manejar el ordenamiento paralelo de listas acopladas lo hace idóneo para esta aplicación educativa.

Documentación de Funciones Principales del Sistema (Parte 1)

El sistema de gestión de notas estudiantiles se compone de un conjunto de funciones modulares para la interacción eficiente con datos de asignaturas y calificaciones. A continuación, se detalla la primera parte de las 13 funciones principales, cubriendo desde la navegación del usuario hasta la manipulación y análisis de datos académicos.

Función	Descripción Técnica
menu()	Muestra 13 opciones de navegación al usuario. Solicita y valida la entrada: un número entero entre 1 y 13. En caso de entrada inválida (no numérica o fuera de rango), informa al usuario y re-muestra el menú. Esta función actúa como el bucle principal del programa, orquestando las llamadas a otras funciones según la selección del usuario.
agregar_asignatura()	<p>Solicita al usuario el nombre de una asignatura y su nota. Realiza dos validaciones:</p> <ul style="list-style-type: none">Validación de Duplicados: Comprueba si la asignatura ya existe en <code>mis_asignaturas</code> (sensible a mayúsculas/minúsculas, ignora espacios extra). Notifica si es duplicada y evita la adición.Validación de Rango de Notas: Asegura que la nota sea un valor numérico entre 0 y 100. Solicita reingreso si es inválida. <p>Tras la validación, la asignatura y su nota se almacenan en las listas paralelas <code>mis_asignaturas</code> y <code>mis_notas</code>, manteniendo su correspondencia por índice.</p>
listar_asignaturas()	Itera sobre <code>mis_asignaturas</code> y <code>mis_notas</code> para mostrar cada asignatura y su nota asociada en formato numerado. Si ambas listas están vacías, informa que no hay asignaturas registradas. El objetivo es proporcionar una visualización rápida y legible del estado actual de las calificaciones.
promedio()	Calcula la media aritmética de las notas en <code>mis_notas</code> . Verifica si <code>mis_notas</code> está vacía; si lo está, retorna 0. En caso contrario, suma todos los valores y los divide por el conteo total de notas. El resultado se presenta redondeado a dos decimales, ofreciendo una métrica consolidada del rendimiento académico general.
aprobadas_reprobadas()	Clasifica las asignaturas según un umbral de aprobación (nota \geq 61 para "aprobada", $<$ 61 para "reprobada"). Itera a través de <code>mis_notas</code> , compara cada nota con el umbral y lista las asignaturas correspondientes. También proporciona un conteo total de asignaturas aprobadas y reprobadas. Es una herramienta clave para el análisis de rendimiento a nivel de asignatura.
buscar_asignatura()	Localiza una asignatura específica mediante una búsqueda lineal en <code>mis_asignaturas</code> . Compara el término de búsqueda (insensible a mayúsculas/minúsculas) con cada nombre de asignatura. Si encuentra una coincidencia, muestra el nombre y la nota asociada. Si la asignatura no se encuentra, notifica al usuario. Para el volumen de datos esperado, su complejidad $O(n)$ es eficiente.
cambiar_nota()	Actualiza la nota de una asignatura existente. Solicita el nombre de la asignatura (buscando de forma similar a <code>buscar_asignatura()</code>). Si la asignatura se encuentra, solicita la nueva nota, la cual se valida en el rango 0-100. Una vez validada, reemplaza la nota antigua en <code>mis_notas</code> por la nueva. Notifica al usuario si la asignatura no es encontrada.

Documentación de Funciones (Parte 2)

Función	Descripción
borrar_asignatura()	Elimina una asignatura y su nota asociada de <code>mis_asignaturas</code> y <code>mis_notas</code> . Requiere confirmación del usuario para evitar eliminaciones accidentales. La localización de la asignatura se realiza por nombre (mediante búsqueda lineal). Es fundamental gestionar los índices de ambas listas para mantener la integridad de los datos. Esta operación afecta el promedio general, requiriendo que las funciones dependientes actualicen sus estados.
ordenar_por_nota()	Ordena las asignaturas y sus notas en <code>mis_asignaturas</code> y <code>mis_notas</code> , respectivamente, utilizando el algoritmo de ordenamiento de burbuja. El criterio es la nota de la asignatura (valores numéricos). Su complejidad temporal es $O(n^2)$, lo que la hace ineficiente para grandes volúmenes de datos. Compara elementos adyacentes para "burbujear" los elementos a sus posiciones correctas.
ordenar_por_nombre()	Ordena las asignaturas y sus notas en <code>mis_asignaturas</code> y <code>mis_notas</code> , respectivamente, usando el algoritmo de burbuja. El criterio es el nombre de la asignatura (cadenas de caracteres). Se debe considerar la sensibilidad a mayúsculas/minúsculas y la codificación. Su complejidad temporal es $O(n^2)$. Genera una lista ordenada por nombre, pre-requisito para la búsqueda binaria.
buscar_binaria()	Realiza una búsqueda binaria para encontrar una asignatura por su nombre. Requiere que <code>mis_asignaturas</code> esté previamente ordenada por nombre. Su complejidad temporal es $O(\log n)$, significativamente más eficiente que la búsqueda lineal para listas largas. Divide repetidamente la porción de la lista en búsqueda hasta encontrar el elemento o determinar su ausencia.
simular_solicitudes()	Simula el procesamiento de solicitudes utilizando una estructura de datos tipo cola (FIFO). Las solicitudes se encolan y desencolan en el orden de llegada. Permite observar el comportamiento del sistema bajo carga, analizando latencia, rendimiento y posibles cuellos de botella. Puede generar un número aleatorio de solicitudes y procesarlas con un tiempo de servicio variable.
ver_historial()	Muestra un registro cronológico de los cambios recientes en asignaturas/notas, gestionado mediante una pila (LIFO). La última acción registrada es la primera en mostrarse, útil para funciones de "deshacer" o auditoría. El historial puede incluir el tipo de cambio, asignatura afectada, valores anteriores/nuevos y marca de tiempo.



Diagrama General del Sistema

Figura 1: Diagrama general del sistema

El diagrama de la **Figura 1** ilustra la arquitectura del sistema de gestión académica, detallando la interacción entre sus componentes clave. El diseño se enfoca en la claridad y la separación de responsabilidades.

El **menú principal** sirve como interfaz primaria, dirigiendo las acciones del usuario hacia las funciones de procesamiento apropiadas.

Las **estructuras de datos** organizan y manipulan la información, utilizando tres tipos principales:

- **Listas:** Implementadas con arreglos dinámicos o listas enlazadas para gestionar objetos `Asignatura` (nombre, nota). Las funciones `añadir_asignatura()`, `borrar_asignatura()`, `ordenar_por_nota()`, `ordenar_por_nombre()` y `buscar_binaria()` interactúan con esta estructura para operaciones CRUD y búsquedas/ordenamientos. La búsqueda binaria requiere que la lista esté ordenada ($O(\log n)$), mientras que las operaciones de ordenamiento por burbuja tienen una complejidad de $O(n^2)$.
- **Colas (FIFO - First-In, First-Out):** Utilizadas por `simular_solicitudes()` para procesar solicitudes (ej. inscripciones) en estricto orden de llegada. Las operaciones de encolar y desencolar tienen una complejidad de $O(1)$.
- **Pilas (LIFO - Last-In, First-Out):** Empleadas para el seguimiento del historial de cambios, como por `ver_historial()`, mostrando las acciones más recientes primero. Cada modificación se "empuja" (push) a la pila ($O(1)$) y se "saca" (pop) para su visualización ($O(1)$). Útil para funcionalidades de "deshacer".

Las **funciones de procesamiento** encapsulan la lógica de negocio, operando sobre estas estructuras de datos bajo el principio de responsabilidad única. Esto mejora la modularidad, legibilidad y facilita la depuración y escalabilidad del sistema.

El **flujo de datos** comienza con la interacción del usuario a través del menú, invocando una función de procesamiento que accede o modifica las estructuras de datos. Los resultados se presentan al usuario, manteniendo la coherencia de la información.

En resumen, la arquitectura propuesta, con su clara distinción entre interfaz, estructuras de datos y lógica de procesamiento, promueve un sistema robusto, fácil de entender y flexible para futuras expansiones.

Pseudocódigo Principal: Control del Flujo de la Aplicación

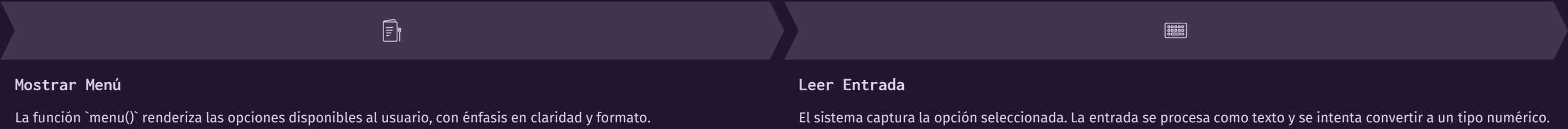
Este bucle principal gestiona la interacción del usuario y el flujo de ejecución, siguiendo una lógica fundamental. Incorpora validación de entrada y manejo de errores para robustez, promoviendo modularidad y extensibilidad del sistema.

```
MIENTRAS Verdadero:      MOSTRAR menu()      LEER opcion      SI opcion NO ES numero:      MOSTRAR "Error: Entrada inválida. Introduce un valor numérico."      SINO SI opcion NO ESTA EN opciones_validas:      MOSTRAR "Error: Opción fuera de rango. Selecciona una opción listada."      SINO:      resultado = EJECUTAR funcion_asociada(opcion)      SI resultado ES Falso:      SALIR      FIN SI      FIN MIENTRAS
```

Este pseudocódigo representa el control de flujo central de la aplicación. Cada componente asegura la operatividad y una experiencia de usuario guiada:

- **MIENTRAS Verdadero:** Bucle infinito para interacción continua hasta una salida explícita.
- **MOSTRAR menu():** Invoca la función ``menu()`` que presenta las opciones disponibles. Retorna la interfaz de opciones, sin procesar la entrada.
- **LEER opcion:** Captura la entrada del usuario. En una implementación real, esto implica la lectura de una cadena y su posterior intento de conversión a tipo numérico (ej. ``int()``).
- **SI opcion NO ES numero:** Primera validación de tipo. Verifica que la entrada sea numérica para prevenir errores de conversión o lógicos.
- **SINO SI opcion NO ESTA EN opciones_validas:** Segunda validación de rango. Comprueba que la opción numérica esté entre las ``opciones_validas`` definidas, evitando la ejecución de funciones inexistentes. ``opciones_validas`` sería una estructura de datos (ej., lista o conjunto) que mapea los índices del menú a las funciones.
- **EJECUTAR funcion_asociada(opcion):** Si la opción es válida, invoca la función correspondiente. La implementación podría usar un ``switch-case`` o un diccionario de funciones para un despacho eficiente $O(1)$.
- **SI resultado ES Falso:** Condición de salida controlada. Una función de menú (ej., "Salir") devuelve ``Falso`` para terminar el bucle de forma limpia, liberando recursos si es necesario.

El diagrama de flujo a continuación detalla las etapas clave de este bucle principal:



Conclusiones Técnicas: Análisis Profundo del Gestor de Notas Simple

Fortalezas Fundamentales del Sistema

- Arquitectura modular y mantenible:** Diseño basado en la separación de responsabilidades (estudiantes, cursos, notas, UI). Permite alta cohesión y bajo acoplamiento, facilitando depuración, mantenimiento y la integración/modificación de funcionalidades sin impacto en el sistema.
- Uso eficiente de estructuras de datos:** Implementa diccionarios (hash maps) para acceso $O(1)$ a estudiantes y cursos por ID, y listas para datos ordenados. Optimiza operaciones de búsqueda, inserción y eliminación, asegurando rendimiento escalable.
- Validación robusta de entradas:** Validación exhaustiva de tipo de dato, rango (ej. notas 0-100) y formato (ej. IDs únicos). Clave para la integridad de datos, prevención de errores de usuario y estabilidad del sistema.
- Historial completo de operaciones:** Mantiene un registro en memoria de operaciones clave, esencial para depuración, auditoría y futuras funcionalidades de deshacer/rehacer.
- Algoritmos apropiados para el contexto:** Utiliza algoritmos eficientes (clasificación, búsqueda, agregación) optimizados para datos académicos simples, equilibrando rendimiento y claridad del código sin complejidades innecesarias.

Características Técnicas Clave

- Implementación en Python puro:** La elección de Python facilita la legibilidad del código, agiliza el desarrollo y aprovecha su vasta biblioteca estándar, permitiendo un enfoque directo en la lógica de negocio.
- Interfaz de consola intuitiva:** UI basada en texto con menús numerados y prompts claros, diseñada para la simplicidad y facilidad de uso, minimizando la curva de aprendizaje.
- Gestión de memoria eficiente:** Python maneja la memoria efectivamente con recolección de basura automática. El diseño del programa evita la carga excesiva de datos, procesando información iterativamente o bajo demanda para un consumo razonable de recursos.
- Código documentado y legible:** Comentarios detallados y docstrings siguiendo PEP 8 garantizan la comprensión del código, facilitan la colaboración y el mantenimiento futuro.
- Escalable para futuras mejoras:** La arquitectura permite una expansión sencilla, incluyendo la adición de nuevas entidades (profesores, aulas), esquemas de calificación o la integración con bases de datos, sin reescritura de la lógica central.

El **Gestor de Notas Simple** se establece como una solución técnica integral para la gestión académica básica. Su diseño prioriza la simplicidad operativa para el usuario y una implementación robusta y flexible a nivel de código.

Arquitectónicamente, la modularidad garantiza **mantenibilidad** y **extensibilidad**. La selección de estructuras de datos y algoritmos demuestra eficiencia para los casos de uso previstos, mientras que la validación rigurosa de entradas asegura la **fiabilidad** del sistema y la integridad de la información.

Para futuras mejoras, la **persistencia de datos** podría implementarse con archivos JSON/CSV o bases de datos como SQLite/PostgreSQL. Una **interfaz gráfica de usuario** (GUI) podría desarrollarse con Tkinter, PyQt, o una interfaz web con Flask/Django. Finalmente, la **integración con sistemas externos** (LMS, SIS) vía APIs transformaría el gestor en una herramienta más potente y conectada.