Problemas Semana 12

Algoritmos Recursivos Avanzados, Integración de Contenidos y Preparación del Proyecto Final

Introducción

Los problemas de esta semana cierran el ciclo de contenidos del curso. Ponen en práctica los dos paradigmas algorítmicos de la clase teórica —**divide y vencerás** y **backtracking**— y, sobre esa base, integran prácticamente todo lo construido a lo largo de las once semanas previas: estructuras de datos compuestas, algoritmos de búsqueda y ordenamiento, recursividad, manejo de archivos de texto y binarios, y medición empírica de rendimiento.

Cada problema sigue requiriendo la aplicación rigurosa de la **metodología de Pólya** (Análisis → Diseño → Codificación → Evaluación) y el **modelo de programa modular** consolidado desde la Semana 4: cada función incluye su sección declarativa (docstring con precondición, postcondición y, cuando corresponda, efectos secundarios), y la sección algorítmica se organiza en Prólogo / Resolución / Epílogo.

Los seis ítems están organizados en tres bloques de problemas más una actividad de cierre, con **conexiones deliberadas**: el Bloque A (Problemas 1 y 2) implementa y compara experimentalmente los algoritmos de ordenamiento por divide y vencerás; el Bloque B (Problemas 3 y 4) ejercita el backtracking en dos aplicaciones clásicas; el Bloque C (Problema 5) plantea un problema integrador complejo, de resolución en equipo, que demanda múltiples estructuras y algoritmos —incluyendo entrada/salida de archivos binarios—; y la Actividad 6 inicia formalmente la preparación del proyecto final integrador de las Semanas 13 a 16.

Recordatorio: Modelo de programa modular — *Cada función incluye su sección declarativa como docstring (descripción, precondición, postcondición, efectos secundarios si los hay), accesible mediante help(). La sección algorítmica se organiza en Prólogo / Resolución / Epílogo. Se mantienen las convenciones del curso: for para iteración con cantidad conocida de pasos y while para iteración de cantidad indeterminada o condición compuesta; sin break (se usan condiciones compuestas y post-inspección del bucle); n >> 1 en lugar de n // 2 cuando se divide por dos; random.seed(42) para reproducibilidad en todo experimento aleatorio; y el uso del context manager with para todo manejo de archivos.*

Sobre el uso de IA generativa en estos problemas — *La clase práctica incluye una discusión colaborativa sobre el uso de IA generativa en la resolución de los problemas del día. Se recomienda que cada estudiante o equipo registre las interacciones significativas con herramientas de IA (el prompt usado, la respuesta obtenida, qué se tomó y qué se descartó, y qué se aprendió). Este registro será insumo del foro de reflexión de la semana y una práctica que continúa durante todo el proyecto final.*

Bloque A — Divide y vencerás: implementación y benchmarking

Problema 1: Merge sort, quick sort y sorted(): comparación empírica

Enunciado

Implementar merge sort y quick sort según el modelo de la clase teórica, y diseñar un experimento que compare empíricamente su rendimiento contra la función nativa `sorted()` de Python.

Implementar las siguientes funciones:

(a) `merge_sort(secuencia)` y su función auxiliar privada `_fusionar(izq, der)`, que devuelva una nueva lista ordenada en forma no decreciente sin modificar la entrada. La docstring debe declarar precondición, postcondición y la complejidad O(n log n) en tiempo y O(n) en espacio auxiliar.

(b) `quick_sort(secuencia)` en la variante no in-place (que construye las sublistas de menores, iguales y mayores), eligiendo como pivote el elemento de la posición media. La docstring debe declarar la complejidad O(n log n) en promedio y O(n²) en el peor caso.

(c) `medir_tiempo(funcion, datos, repeticiones=3)` que ejecute la función de ordenamiento sobre una copia de los datos tantas veces como indique `repeticiones` y devuelva el menor tiempo medido con `time.perf_counter()`. Usar el mínimo y no el promedio, por ser menos sensible al ruido del sistema operativo.

(d) `comparar_ordenamientos(tamanios, patron)` que, para cada tamaño n de la lista `tamanios`, genere datos según el patrón indicado ("aleatorio", "ordenado" u "ordenado inverso"), mida los tres algoritmos e imprima una tabla comparativa. Fijar `random.seed(42)` al inicio para reproducibilidad.

Ejecutar el experimento con tamaños 1 000, 10 000 y 100 000, sobre los tres patrones de datos. Registrar los resultados en una tabla y responder por escrito: (1) ¿cuántas veces más rápido es `sorted()` que las implementaciones propias, y por qué, si los tres algoritmos son O(n log n)? (2) Al pasar de n a 10·n, ¿el tiempo se multiplica por 10? ¿Es consistente con O(n log n)? (3) ¿Cómo se comporta quick sort sobre datos ya ordenados con la regla de pivote elegida?

Orientaciones para la resolución

Análisis: Los tres algoritmos comparten complejidad asintótica O(n log n), de modo que el experimento no busca distinguir clases de complejidad sino medir las constantes ocultas. La hipótesis es que sorted() —implementado en C dentro del intérprete— será notablemente más rápido que las implementaciones en Python puro, sin que ello signifique que sea asintóticamente superior.

Diseño: Definir medir_tiempo para que trabaje siempre sobre una copia fresca de los datos (list(datos)): merge_sort y quick_sort devuelven una lista nueva, pero sorted() también es no destructiva, y se busca medir las tres en igualdad de condiciones. Para el patrón "ordenado" usar list(range(n)); para "ordenado inverso", list(range(n, 0, -1)); para "aleatorio", una lista por comprensión con random.randint.

Evaluación: Verificar primero la corrección: para una lista de prueba conocida, las tres funciones deben producir exactamente el mismo resultado que sorted(). Sólo después de validar la corrección tiene sentido medir tiempos. Para confirmar O(n log n), calcular el cociente entre el tiempo de 10·n y el de n: debería rondar 13–15, no 10, porque 10·n·log(10n) / (n·log n) es mayor que 10.

> *Conexión con la Semana 6 — En la Semana 6 se presentaron merge sort y quick sort de manera introductoria, junto con los algoritmos cuadráticos (selección, inserción, burbuja). Este problema los retoma con implementación completa y análisis riguroso de complejidad. La medición empírica con time es la misma herramienta introducida en la Semana 5 para comparar búsqueda lineal y binaria.*

Problema 2: Quick sort y la elección del pivote: provocando el peor caso

Enunciado

Investigar experimentalmente cómo la regla de elección del pivote determina si quick sort se comporta en O(n log n) o degrada a O(n²).

(a) Implementar tres variantes de quick sort que se diferencien únicamente en la elección del pivote: `quick_sort_primero` (pivote = primer elemento), `quick_sort_medio` (pivote = elemento de la posición media) y `quick_sort_aleatorio` (pivote = elemento en una posición elegida con random.randint). Las tres comparten la misma estructura de partición.

(b) Instrumentar cada variante con un contador de la **profundidad máxima de recursión** alcanzada. La profundidad se pasa como parámetro adicional que se incrementa en cada llamada recursiva; la función conserva el máximo observado. Una profundidad cercana a $\log_2(n)$ indica buen balance; una profundidad cercana a n indica el peor caso.

(c) Ejecutar las tres variantes sobre tres patrones de entrada de tamaño 2 000: lista aleatoria, lista ya ordenada y lista ordenada en sentido inverso. Construir una tabla de 3×3 con las profundidades máximas observadas.

(d) Analizar por escrito: ¿qué combinación de variante y patrón produce el peor caso O(n²)? ¿Por qué la regla "primer elemento" es especialmente peligrosa con datos del mundo real, que con frecuencia ya están parcial o totalmente ordenados? ¿Qué ventaja ofrece el pivote aleatorio frente a entradas construidas adversariales?

Orientaciones para la resolución

Análisis: El peor caso de quick sort se produce cuando el pivote divide el problema de la manera más desbalanceada posible: una sublista vacía y otra con n-1 elementos. En esa situación la recursión profundiza n veces en lugar de log n. El experimento busca exhibir empíricamente esta degradación midiendo la profundidad de la pila de llamadas.

Diseño: Para medir la profundidad sin alterar la lógica, agregar un parámetro profundidad con valor inicial 0 y una estructura mutable (por ejemplo, una lista de un elemento) que conserve el máximo observado, accesible a través de las llamadas recursivas. Importante: con la lista ya ordenada de tamaño 2 000, la variante quick_sort_primero puede superar el límite de recursión

de Python (sys.getrecursionlimit(), típicamente 1 000); se puede elevar transitoriamente con sys.setrecursionlimit o reducir el tamaño de prueba a 900 y dejar constancia del fenómeno.

Evaluación: La tabla debe mostrar con claridad que quick_sort_primero sobre datos ordenados (directa o inversamente) alcanza profundidad ≈ n, mientras que quick_sort_medio y quick_sort_aleatorio se mantienen cerca de $\log_2(2000) \approx 11$. El propio RecursionError, si aparece, es evidencia experimental válida del peor caso y debe documentarse, no ocultarse.

> *Conexión con la Semana 9 — La profundidad de recursión que aquí se mide es la altura de la pila de ejecución estudiada en la Semana 9. El RecursionError no es un error de programación sino el síntoma observable de que el peor caso $O(n^2)$ agotó el espacio de pila reservado por el sistema operativo para el proceso.*

Bloque B — Backtracking

Problema 3: Generación de permutaciones con traza de la exploración

Enunciado

Implementar la generación de todas las permutaciones de una lista mediante backtracking, e instrumentar el algoritmo para producir una traza de la exploración del árbol de búsqueda.

(a) Implementar `permutaciones(elementos)` y su función recursiva auxiliar `_explorar_permutacion(elementos, permutacion, usados, soluciones)`, siguiendo el modelo de la clase teórica con parámetros explícitos (sin closures). La función debe devolver una lista con las n! permutaciones, cada una como una lista.

(b) Verificar que al registrar cada solución se guarde una **copia** (`list(permutacion)`) y no la referencia. Documentar en la docstring por qué guardar la referencia produciría una lista de soluciones todas iguales y vacías.

(c) Instrumentar el algoritmo para que, además de acumular las soluciones, imprima una **traza** de cada acción significativa: "aplicar X", "registrar solución" y "back-track (pop X)". Cada línea de traza debe mostrar el estado actual de `permutacion` y de `usados`. Indentar la salida según la profundidad de la recursión para que el árbol se visualice en la consola.

(d) Ejecutar para `permutaciones([1, 2, 3])` y verificar que la traza recorre las 6 hojas en orden lexicográfico. Contar las llamadas recursivas totales y comparar con el tamaño del árbol de búsqueda (1 raíz + 3 + 6 + 6 = 16 nodos).

Orientaciones para la resolución

Análisis: Una permutación se construye como una elección secuencial: primero el elemento de la primera posición, luego el de la segunda, y así sucesivamente. El espacio de búsqueda es un árbol de profundidad n con factor de ramificación decreciente (n, n-1, n-2, ..., 1), de modo que tiene n! hojas. La lista usados es la estructura que implementa la poda "no repetir un elemento ya incluido".

Diseño: La traza indentada se logra pasando la profundidad como parámetro y construyendo el prefijo de espacios con `"  " *` profundidad. El par append/pop sobre permutacion, junto con la asignación True/False sobre usados, materializa la simetría hacer/deshacer del backtracking: toda decisión aplicada antes de la llamada recursiva debe deshacerse después.

Evaluación: La traza correcta muestra, después de cada solución registrada, una cascada de back-tracks consecutivos que desenrolla la recursión hasta encontrar una decisión alternativa pendiente. Al terminar la ejecución, tanto permutacion como usados deben haber vuelto exactamente a su estado inicial ([] y [False, False, False]): el algoritmo es no destructivo respecto de su propio estado de trabajo.

> *Conexión con la Semana 10 — La lista usados es una estructura auxiliar paralela a elementos. La elección de representar el estado parcial como una lista (mutable, con append y pop en O(1) al final) es una decisión de diseño que se apoya en el repertorio de estructuras de la Semana 10. El mismo problema podría resolverse con otras representaciones, con distinto costo.*

Problema 4: Resolución de un laberinto simple por backtracking

Enunciado

Implementar la búsqueda de un camino entre dos celdas de un laberinto mediante backtracking, modelando el laberinto como una búsqueda en profundidad sobre un grafo implícito.

(a) Implementar `resolver_laberinto(laberinto, inicio, destino)` y su función recursiva auxiliar con parámetros explícitos. El laberinto es una lista de listas de 0 (celda libre) y 1 (pared); `inicio` y `destino` son tuplas (fila, columna). La función devuelve la lista de celdas del camino, o `None` si no existe camino.

(b) Usar un **conjunto** (`set`) para las celdas visitadas. Justificar en la docstring por qué un conjunto es preferible a una lista para esta función, en términos del costo de la operación de pertenencia.

(c) Probar la función con el laberinto 5×5 del documento teórico y con al menos un laberinto sin solución (destino encerrado por paredes). Verificar que en el segundo caso la función devuelve `None` tras explorar exhaustivamente el espacio alcanzable.

(d) Implementar `imprimir_laberinto(laberinto, camino)` que muestre el laberinto en consola con caracteres: pared, celda libre, y las celdas del camino encontrado marcadas de manera distinguible. Analizar por escrito: ¿el camino encontrado es el más corto? ¿De qué depende? ¿Qué pasaría si se cambiara el orden en que se prueban los movimientos cardinales?

Orientaciones para la resolución

Análisis: El laberinto es un grafo implícito: cada celda libre es un nodo y cada par de celdas libres adyacentes es una arista. Resolver el laberinto por backtracking equivale a una búsqueda en profundidad (DFS) sobre ese grafo, con marcado de visitados para evitar ciclos. El estado parcial es la celda actual más el conjunto de celdas ya visitadas.

Diseño: Definir la lista de movimientos cardinales como una lista de pares (df, dc) —por ejemplo, [(-1,0), (1,0), (0,-1), (0,1)] para Norte, Sur, Oeste, Este—. La poda combina tres condiciones en un único if con and: estar dentro de los límites del laberinto, que la celda destino no sea pared, y que no haya sido visitada. La pertenencia siguiente not in visitadas es O(1) promedio en un conjunto y O(n) en una lista.

Evaluación: El camino devuelto es el primero que la búsqueda en profundidad encuentra, y no necesariamente el más corto: depende del orden en que se prueban los movimientos. Cambiar ese orden cambia el camino. Para el laberinto sin solución, la función debe terminar (no quedar en bucle infinito), lo que confirma que el marcado de visitados evita revisitar celdas.

Conexión con la Semana 11 y hacia el proyecto — La búsqueda en profundidad sobre un grafo implícito es un patrón que reaparece en numerosos problemas: recorridos de árboles, exploración de estados, resolución de puzzles. Para el proyecto final, reconocer cuándo un problema tiene forma de "búsqueda en un espacio de estados" habilita aplicar este patrón.

Bloque C — Problema integrador (resolución en equipo)

El Problema 5 es un **problema integrador complejo**, de resolución en equipo durante la clase práctica, que demanda combinar múltiples estructuras de datos y algoritmos en un único programa. A diferencia de los problemas de los bloques anteriores —cada uno enfocado en una técnica— este problema exige decidir **qué técnica aplicar a cada parte** y articular las partes en una solución modular coherente. Funciona como ensayo general del proyecto final: el equipo enfrenta, en pequeña escala y con el enunciado ya dado, el mismo tipo de trabajo de integración que deberá hacer en las Semanas 13 a 16 sobre un problema de su propia elección.

Problema 5: Sistema de turnos de un consultorio médico

Enunciado

Desarrollar, en equipos de 2 o 3 integrantes, un sistema de gestión de turnos para un consultorio médico. El sistema persiste los datos en un **archivo binario de registros de longitud fija**, mantiene índices en memoria mediante **diccionarios**, produce reportes **ordenados**, resuelve **búsquedas** eficientes por distintos criterios, y asigna turnos a una agenda diaria mediante **backtracking**. El problema se descompone en cuatro módulos; se sugiere repartir su desarrollo entre los integrantes del equipo y luego integrarlos.

Módulo 1 — Persistencia binaria de pacientes

Diseñar el registro de paciente de longitud fija y sus operaciones de lectura y escritura. El registro tiene la siguiente estructura:

Campo	Tipo struct	Tamaño
dni	i (int32)	4 bytes
apellido	30s	30 bytes
nombre	24s	24 bytes

Campo	Tipo struct	Tamaño
telefono	16s	16 bytes
prioridad	B (uint8)	1 byte

(a) Declarar `FORMATO = '<i30s24s16sB'` y `TAM_REGISTRO = struct.calcsize(FORMATO)` como constantes globales (fuente única de verdad). Implementar `empaquetar_paciente` y `desempaquetar_paciente`, con codificación UTF-8, truncado de cadenas largas y removido del relleno de ceros al desempaquetar. El campo `prioridad` es un entero de 1 (alta) a 3 (baja).

(b) Implementar `crear_archivo_pacientes(ruta, lista_pacientes)` y `leer_paciente(archivo, k)` con acceso directo por offset (`seek(k * TAM_REGISTRO)`). Usar el context manager `with` en todo acceso a archivo.

Módulo 2 — Índices en memoria

(c) Implementar `construir_indices(ruta)` que recorra una sola vez el archivo binario y devuelva dos diccionarios: `indice_por_dni` (clave: DNI, valor: posición k del registro en el archivo) e `indice_por_apellido` (clave: apellido, valor: lista de posiciones, porque puede haber apellidos repetidos).

(d) Implementar `buscar_por_dni(archivo, indice_por_dni, dni)` que resuelva la búsqueda en O(1) promedio consultando el diccionario y leyendo un único registro. Comparar conceptualmente, en la docstring, con el costo de una búsqueda secuencial O(n) sobre el archivo sin índice.

Módulo 3 — Reportes ordenados

(e) Implementar `listar_pacientes_ordenados(ruta, criterio)` que lea todos los pacientes del archivo y devuelva la lista ordenada según el criterio indicado: "apellido" (alfabético) o "prioridad" (de 1 a 3, y dentro de cada prioridad, por apellido). Reutilizar la implementación de `merge_sort` del Problema 1 —que es estable— para el ordenamiento por prioridad con desempate por apellido.

(f) Justificar por escrito por qué la estabilidad del algoritmo de ordenamiento es relevante para el criterio "prioridad": describir el procedimiento de dos pasadas (ordenar por apellido y luego por prioridad) y explicar qué se rompería si el segundo ordenamiento no fuera estable.

Módulo 4 — Asignación de la agenda diaria por backtracking

El consultorio tiene una agenda de **franjas horarias** (por ejemplo, 8 franjas de 30 minutos). Algunos pacientes tienen **restricciones de disponibilidad**: una lista de las franjas en las que cada uno puede asistir. El problema es asignar cada paciente de una lista del día a una franja, respetando que (1) cada franja recibe a lo sumo un paciente y (2) cada paciente queda en una franja compatible con su disponibilidad.

(g) Implementar `asignar_agenda(pacientes_del_dia, franjas, disponibilidad)` mediante backtracking. El estado parcial es la asignación construida hasta el momento (un diccionario franja → paciente, o paciente → franja). La poda descarta asignar un paciente a

una franja ya ocupada o no disponible para él. La función devuelve una asignación válida, o `None` si no existe ninguna.

(h) Probar con un caso que tenga solución y con un caso sobre-restringido que no la tenga (más pacientes que franjas compatibles). Para el caso con solución, verificar que la asignación devuelta respeta todas las restricciones. Discutir: ¿cuántas asignaciones posibles habría que revisar por fuerza bruta, y cuántas evita la poda?

Integración

(i) Escribir un programa principal que articule los cuatro módulos en un flujo completo: crear el archivo de pacientes, construir los índices, ofrecer un menú de consulta (buscar por DNI, listar ordenado por apellido o prioridad) y resolver la agenda del día. El programa principal no debe contener lógica de los módulos: sólo coordinarlos. Cada módulo se entrega con sus funciones documentadas y con sus propios casos de prueba.

Orientaciones para la resolución

Análisis: El problema no es "un" problema sino cuatro subproblemas de naturaleza distinta — persistencia binaria, indexación, ordenamiento y búsqueda combinatoria— que comparten un mismo dominio. El trabajo de análisis central es reconocer qué técnica del curso resuelve cada subproblema: archivos binarios de longitud fija para la persistencia (Semana 8), diccionarios para los índices (Semana 10), merge sort estable para los reportes (Semanas 6 y 12), y backtracking para la agenda (Semana 12).

Diseño: La descomposición en módulos es a la vez una estrategia de diseño y una estrategia de trabajo en equipo: cada integrante puede tomar uno o dos módulos y avanzar en paralelo, siempre que las interfaces entre módulos —qué recibe y qué devuelve cada función— estén acordadas de antemano. Conviene escribir primero las firmas de las funciones con sus docstrings (precondición y postcondición) y recién después codificar los cuerpos. El programa principal de integración debe escribirse al final, cuando los módulos ya pasan sus pruebas individuales.

Evaluación: Cada módulo se valida por separado con sus propios casos antes de integrar. Para el Módulo 1, verificar que os.path.getsize(ruta) coincide con cantidad_de_pacientes * TAM_REGISTRO. Para el Módulo 2, comprobar que toda búsqueda por DNI devuelve el registro correcto. Para el Módulo 3, contrastar el resultado contra sorted() con la misma clave. Para el Módulo 4, verificar manualmente que la asignación devuelta respeta disponibilidad y unicidad de franja. La integración recién se prueba cuando los cuatro módulos están validados.

Conexión integradora — Este problema reúne, en un único programa, contenidos de las Semanas 6, 8, 10 y 12. Es deliberadamente un ensayo general del proyecto final: la misma exigencia de descomponer un problema realista en módulos, repartir el trabajo, acordar interfaces e integrar. La diferencia es que aquí el enunciado está dado; en el proyecto, el equipo elegirá su propio problema.

Actividad de cierre — Arranque del proyecto final

Esta actividad no es un problema de programación: es el **punto de partida formal del proyecto final integrador** que ocupará las Semanas 13 a 16. Se realiza en la última parte de la clase práctica y su producto —el plan preliminar del proyecto— se completa como parte del trabajo de extracursado y se entrega al final de la semana.

Parte 1 — Formación de equipos

Conformar equipos de 2 o 3 integrantes. Los equipos son estables para todo el proyecto. Al formarse, cada equipo registra: nombre de los integrantes, un medio de comunicación acordado y la creación de un repositorio compartido (por ejemplo, en GitHub) donde versionar el trabajo. El historial de commits formará parte de la evaluación del proyecto, no como métrica de cantidad sino como evidencia del proceso y de la contribución de cada integrante.

Parte 2 — Selección del problema

Cada equipo elige el problema que abordará. No hay un enunciado único: la elección es parte del trabajo. Un buen problema para el proyecto final cumple, idealmente, los siguientes criterios:

Criterio	Pregunta que el equipo debe poder responder
Tractabilidad	¿Es realista resolverlo en 4 semanas con las herramientas del curso?
Riqueza	¿Demanda al menos 3 o 4 contenidos diferentes del curso, no uno solo?
Modularidad	¿Se descompone en piezas asignables a distintos integrantes?
Verificabilidad	¿Se pueden definir casos de análisis claros y comprobables?
Interés	¿Le interesa al equipo? Cuatro semanas son muchas si el problema no engancha.
Especificidad	¿Está acotado? "Hacer una red social" no; "sistema de turnos con archivo binario, búsqueda por DNI y reportes ordenados" sí.

A modo de referencia —y sin que la lista sea obligatoria ni exhaustiva—, son problemas con el alcance adecuado, por ejemplo: un gestor de inventario de un depósito con control de stock y reposición; un sistema de reservas de una biblioteca con archivo binario de ejemplares y préstamos; un planificador de horarios de cursada que asigne materias a franjas evitando superposiciones; o un analizador de un archivo de registros de actividad (logs) que produzca estadísticas y reportes ordenados. Cada equipo puede adoptar uno de estos, una variante, o proponer un problema propio. La pertinencia y el alcance del problema elegido se validan con el docente antes de avanzar.

Parte 3 — Plan preliminar del proyecto

Completar y entregar, como producto de extracursado, un plan preliminar con la siguiente plantilla. El plan no es definitivo —se refinará en las Semanas 13 y 14— pero debe ser concreto y verificable.

Sección del plan	Contenido esperado
Descripción del problema	Qué resuelve el sistema, en 1 o 2 párrafos, con alcance acotado y explícito.
Contenidos del curso involucrados	Lista de los contenidos del curso que el problema demanda (estructuras, archivos, algoritmos), con una frase que justifique cada uno.
Casos de análisis	Al menos 4 o 5 casos —normales y límite— que el sistema deberá manejar correctamente, con la entrada y la salida esperada de cada uno.
Estrategia de diseño	Descomposición preliminar en módulos o funciones principales, y qué algoritmo o estructura resolverá cada parte.
Reparto preliminar del trabajo	Qué integrante se hará cargo, inicialmente, de cada módulo.
Riesgos anticipados	Qué parte del problema parece más difícil o incierta, y una idea de cómo abordarla.

Sobre la rúbrica de evaluación — *La rúbrica completa del proyecto final se presenta y discute en la primera clase de la Semana 13. Sus cinco dimensiones son: pensamiento algorítmico (calidad del análisis, pertinencia de las estructuras y algoritmos, manejo de complejidad); modularización y reuso (funciones con responsabilidad clara, interfaces bien definidas, ausencia de duplicación); documentación interna (docstrings con precondiciones y postcondiciones, casos de análisis documentados); calidad del producto (corrección en casos normales y límite); y ética y reflexión (registro de interacciones con IA, justificación de decisiones, contribución individual evidenciada en el historial de versionado). Conocer estas dimensiones desde ahora permite orientar el plan preliminar.*

Recomendación de estrategia — *Antes de empezar a codificar, conviene dedicar tiempo desproporcionado al análisis y al diseño. Un equipo que llega a la Semana 14 con el algoritmo pseudocodificado, los casos de análisis identificados y los módulos esquematizados, codifica en una semana lo que un equipo sin diseño codifica y recodifica en tres. La tentación de codificar de inmediato es fuerte: conviene resistirla.*

Desafíos extracurriculares grupales

Las siguientes cuatro propuestas profundizan temas tratados conceptualmente en la teoría pero no implementados en clase. Son **opcionales** y de resolución en equipo. A diferencia de la Actividad 6 —que prepara el proyecto final—, estos desafíos son ejercicios algorítmicos acotados para equipos que quieran profundizar.

Desafío 1 — Problema de las N reinas

Implementar, mediante backtracking, la ubicación de N reinas en un tablero de ajedrez de N×N de modo que ninguna ataque a otra (ninguna comparte fila, columna ni diagonal). La función `resolver_n_reinas(n)` devuelve una solución (la posición de cada reina) o `None` si no existe. Implementar también `contar_soluciones(n)` que cuente todas las soluciones distintas.

Medir el tiempo de `contar_soluciones` para n de 4 a 12 y construir una tabla. Analizar cómo crece el costo y por qué la poda por diagonales es decisiva para que el problema sea tratable. Como entregable, un informe breve que relacione el crecimiento observado con la naturaleza combinatoria del problema.

Desafío 2 — Quick sort in-place con partición de Lomuto

Implementar la variante **in-place** de quick sort, que ordena la lista sin construir sublistas nuevas, usando el esquema de partición de Lomuto con intercambios sobre índices. La firma es `quick_sort_inplace(secuencia, bajo, alto)`. Comparar empíricamente su uso de memoria y su tiempo contra la versión no in-place del Problema 1.

Como entregable, un informe que explique la diferencia de complejidad espacial entre ambas variantes —O(log n) de pila en la in-place frente a O(n) de sublistas en la no in-place— y que discuta por qué la versión in-place pierde la estabilidad.

Desafío 3 — Suma de subconjuntos con poda

Dado un conjunto de enteros positivos y un valor objetivo, encontrar mediante backtracking todos los subconjuntos cuya suma sea exactamente el objetivo. Implementar `subconjuntos_suma(numeros, objetivo)` con dos podas: descartar una rama cuando la suma parcial ya supera el objetivo, y descartar cuando ni sumando todos los elementos restantes se alcanza el objetivo.

Como entregable, comparar experimentalmente la cantidad de nodos del árbol de búsqueda explorados con y sin las podas, sobre un mismo conjunto de entrada, y cuantificar cuántas ramas evita cada poda.

Desafío 4 — Divide y vencerás: el par de puntos más cercano

Implementar el algoritmo de divide y vencerás que encuentra, en un conjunto de puntos del plano, el par de puntos a menor distancia, en O(n log n). Implementar también la versión por fuerza bruta O(n²) y verificar que ambas producen el mismo resultado.

Como entregable, un informe que explique cómo se combinan las soluciones de las dos mitades —en particular, el tratamiento de los pares de puntos que quedan a un lado y otro de la línea divisoria— y una comparación empírica de tiempos entre ambas versiones para tamaños crecientes.

Distribución sugerida del tiempo (9 horas)

Ámbito	Problemas / Tareas	Tiempo	Contexto
Clase práctica (3 h)	Problema 1	~40 min	Implementación de merge sort y quick sort; planteo del benchmarking
	Problema 2	~35 min	Variantes de pivote y medición de profundidad de recursión

Ámbito	Problemas / Tareas	Tiempo	Contexto
	Problema 3	~25 min	Permutaciones con traza de la exploración
	Problema 4 (planteo)	~20 min	Backtracking del laberinto; dejar la prueba completa para extracursado
	Problema 5 (arranque)	~40 min	Análisis y descomposición en módulos; reparto entre integrantes
	Actividad 6 (Partes 1 y 2)	~20 min	Formación de equipos y selección preliminar del problema
Extracursado individual / por pares (3 h)	Problemas 1 y 2 (corridas)	~1 h	Ejecutar los experimentos completos y analizar los resultados
	Problema 4 (finalización)	~45 min	Completar la prueba con el laberinto sin solución y la impresión
	Ejercicios de Moodle	~1 h	Ejercicios de backtracking y divide y vencerás
	Lectura y autoevaluación	~15 min	Autoevaluación integral de competencias (semanas 1–12)
Extracursado grupal (6 h)	Problema 5 (desarrollo)	~3 h	Codificación, prueba e integración de los cuatro módulos
	Actividad 6 (Parte 3)	~1 h 30 min	Completar y entregar el plan preliminar del proyecto final
	Foro de reflexión sobre IA	~30 min	Participación en el foro de la semana
	Desafío grupal (opcional)	~1 h	Selección y avance de uno de los cuatro desafíos extracurriculares