Obligatorio 2
Estructuras de Datos y Algoritmos 2

2025

https://github.com/RominaCarsin/ObligatorioAlgoritmos

326152 - Romina Carsin

246326 - Luis Sanguinetti

Docentes

Francisco Bouza

Introducción

En este obligatorio trabajamos los ejercicios del 6 al 10 utilizando una estructura organizada en un único archivo por ejercicio donde se utiliza un main para manejar la entrada, salida y las funciones o estructuras principales según correspondía. Dependiendo del ejercicio se combinaron diferentes estrategias, incluyendo programación dinámica, estructuras equilibradas como AVL, tablas hash con encadenamiento, heaps y técnicas de backtracking. Para el desarrollo de los ejercicios se tomaron como referencia ejemplos vistos en clase, especialmente de las diapositivas y los códigos realizados en conjunto.

En cuanto al uso de IA, esta se utilizó exclusivamente como herramienta de apoyo para depurar errores cuando el debugging tradicional no lograba mostrar el origen del problema, así como para buscar explicaciones alternativas de conceptos nuevos. No se usó para generar soluciones completas sin supervisión; cada sugerencia fue revisada, adaptada y corregida manualmente para asegurar el entendimiento del funcionamiento interno de cada algoritmo. Algunas intervenciones de la IA fueron indicadas en el propio código y se integraron únicamente como complemento del aprendizaje y/o corrección de errores particulares .

Ejercicio 6

Para el ejercicio 6 se solicitó determinar la cantidad mínima de mejoras necesarias para avanzar desde la posición inicial hasta la posición final evitando pozos. Para resolverlo utilizamos una estrategia Greedy basada en un MaxHeap. La estructura principal elegida fue el MaxHeap porque permite mantener en todo momento las mejoras disponibles ordenadas por potencia, garantizando que siempre podamos tomar la mejora más grande cuando el avance quede bloqueado. Además, se recorren las listas de pozos y mejoras de forma lineal mediante dos punteros crecientes, lo que permite procesarlas de manera eficiente sin reescaneos. Tomamos como base para resolverlo los ejemplos realizados en clase de Gready tales como cambio.cpp.

Justificación de la complejidad temporal

INSERCIÓN Y EXTRACCIÓN EN EL HEAP

En cada paso del recorrido se insertan en el heap todas las mejoras cuya posición sea alcanzable. Cada inserción cuesta O(log m), y cada mejora se inserta una única vez porque el puntero avanza secuencialmente. Cuando el jugador no puede continuar, se extrae la mejora de mayor potencia del heap; esta operación también cuesta O(log m). En total, a lo largo de toda la ejecución puede haber a lo sumo m inserciones y m extracciones, por lo que estas operaciones aportan una complejidad total de O(m log m).

RECORRIDO DE POZOS Y MEJORAS

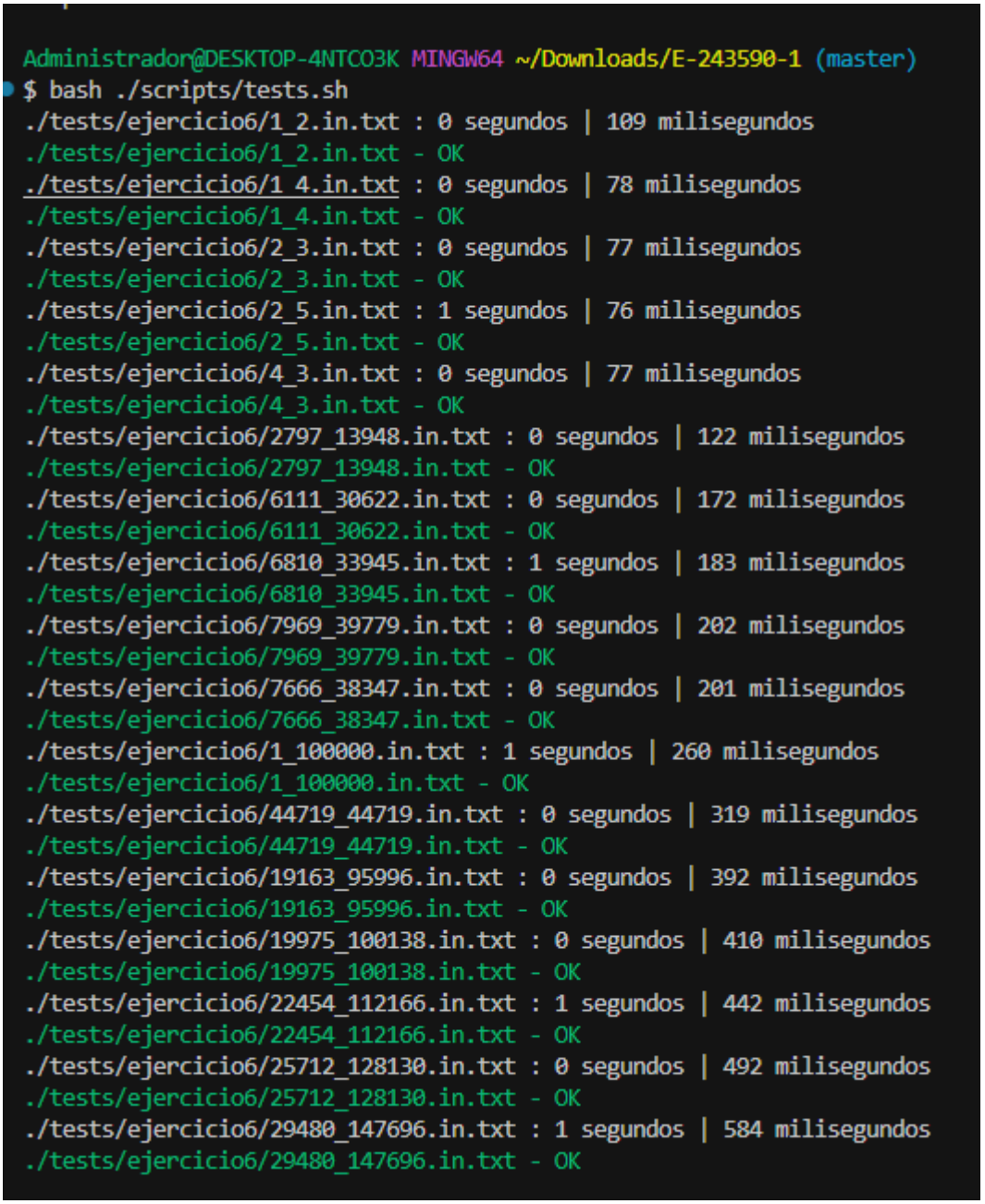
Para manejar pozos y mejoras se emplean los punteros posPozo y posMejora. Ambos avanzan siempre hacia adelante y nunca retroceden. Esto significa que tanto la lista de pozos como la lista de mejoras se recorren una sola vez, aportando una complejidad lineal O(n + m) sin repeticiones innecesarias.

MOVIMIENTO Y CÁLCULO DEL LÍMITE

En cada iteración se calcula el alcance actual sumando la potencia disponible y se determina hasta dónde se puede avanzar antes de caer en un pozo. Esta operación es constante, salvo por la llamada a syncPozos, que simplemente incrementa el puntero hasta descartar pozos ya pasados. Como posPozo solo aumenta, el costo acumulado de este ajuste sigue siendo lineal.

Sumando los componentes previos, se obtiene una complejidad total de O(m log m + n).

Resultados De Prueba



Ejercicio 7

Para poder resolver este ejercicio fue necesario utilizar dos estructuras complementarias: un árbol AVL y un algoritmo MergeSort adaptado para contabilizar inversiones. El AVL se usó como estructura principal para mapear cada clave (string) a su posición original en la primera lista, permitiendo encontrar su índice en tiempo logarítmico. Luego, esa secuencia de índices se procesó mediante un MergeSort modificado para contar inversiones de forma óptima. Se decidió utilizar AVL debido a que cumple búsquedas e inserciones en O(log N) debido a su balanceo automático, mientras que un hash no lo cumple en el peor caso. Se utilizó long long para expandir el alcance del int debido al gran tamaño de los datos trabajados.

Justificación de la complejidad temporal

INSERCIÓN Y BÚSQUEDA EN EL AVL

Cada uno de los elementos de la primera lista se inserta en el AVL junto con su posición. Insertar en un árbol AVL cuesta O(log N) debido a que el árbol se mantiene balanceado y cada inserción requiere como máximo un número constante de rotaciones. Por lo tanto, insertar todos los elementos aporta una complejidad total de O(N log N). Para procesar la segunda lista, cada elemento se busca en el AVL para obtener su posición original O(N log N).

GENERACIÓN DEL ARREGLO Y PREPARACIÓN PARA EL CONTEO

Una vez obtenida la correspondencia entre ambas listas, se forma un arreglo de tamaño N que contiene las posiciones originales de los elementos en el orden de la segunda lista. Esta transformación se realiza en tiempo lineal, O(n).

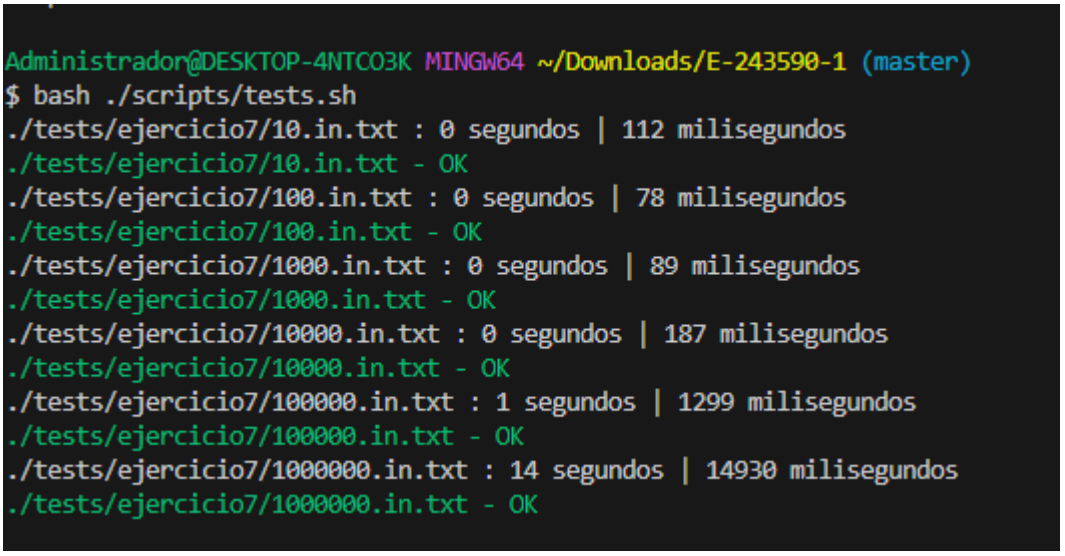
MERGESORT Y CONTEO DE INVERSIONES:

Para contar cuántos pares están invertidos entre ambas listas, se empleó un MergeSort modificado. Este método divide el problema en dos mitades y combina las soluciones detectando cuántos elementos de la derecha deben adelantarse a elementos de la izquierda. Cada mezcla aporta el conteo de inversiones necesarias y opera en tiempo lineal

respecto al tamaño del segmento actual. Como el árbol de llamadas de MergeSort tiene una altura O(log N) y cada nivel procesa en total O(N), la complejidad global del conteo es O(N log N).

Sumando los pasos, se obtiene que el AVL aporta O(N log N) por las inserciones y búsquedas, y el MergeSort aporta O(N log N) por el conteo de inversiones. Por lo tanto, la complejidad total del algoritmo es O(N log N).

Resultados De Prueba



Ejercicio 8

Se resuelve mediante programación dinámica,para implementar una solución eficiente utilizamos una función recursiva con memorización, donde cada estado está determinado por tres parámetros: los límites izquierdo y derecho del intervalo a evaluar, y la cantidad adicional de cristales iguales contiguos a la derecha. Dado que la cantidad de estados es muy grande, fue necesario implementar una tabla hash propia para almacenar estos resultados y así evitar recomputar subproblemas, lo que permite que la solución sea viable dentro de los límites esperados. Se eligió hash ya que maneja una memoria dinámica porque en un principio se había utilizado una matriz de 3 dimensiones pero esta generaba errores en la prueba mas grande con respecto a cantidad de datos. Esta decision de utilizar hash fue antes de que reduzcan la cantidadde pruebas, a pesar de eso pasa todas.

Justificación de la complejidad temporal

MEMORIZACIÓN EN TABLA HASH

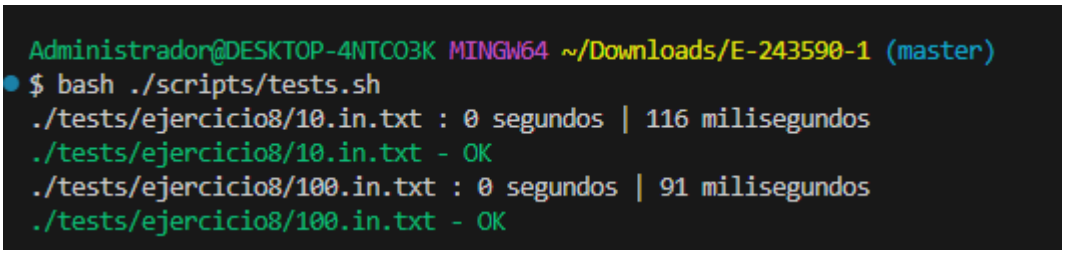
Cada estado (l,r,k) representa una configuración única del intervalo de cristales que estamos evaluando. La tabla hash implementada permite guardar y recuperar valores en tiempo promedio O(1), ya que se accede mediante funciones hash y listas enlazadas para resolver colisiones.

COMPLEJIDAD DE LA TRANSICIÓN

Cada estado realiza como máximo un bucle lineal sobre su intervalo y realiza llamadas recursivas a subintervalos más pequeños. Sin memorización, esto sería exponencial debido a las recombinaciones, pero al almacenar cada resultado la primera vez que se calcula, las transiciones se reducen a O(n) por estado. Así, el costo total viene dado por la cantidad de estados distintos multiplicado por el costo de cada transición.

La complejidad teórica del algoritmo con estas técnicas se ubica en $O(n^3)$, que es la cota estándar para este tipo de problemas (relacionado con la clásica variante de “Remove Boxes”). La tabla hash garantiza recuperación y almacenamiento O(1), lo cual mantiene la complejidad dominada únicamente por la exploración de estados.

Resultados De Prueba



Ejercicio 9

Se utilizó una programación dinámica tridimensional, donde cada estado representa la mejor solución posible considerando los primeros *i* archivos y límites de tamaño y líneas disponibles. Se diseñó una matriz tridimensional mat[i][s][l] que almacena el máximo puntaje alcanzable usando los primeros *i* archivos sin superar los valores de tamaño *s* y líneas *l*.

Justificación de la complejidad temporal

CONSTRUCCIÓN DE LA MATRIZ DE PROGRAMACIÓN DINÁMICA

La solución recorre todos los archivos y, para cada uno, evalúa los posibles valores de tamaño desde 0 hasta S y de líneas desde 0 hasta L. Esto implica recorrer una estructura de tamaño (N × S × L). En cada celda se calculan dos posibilidades: no tomar el archivo actual o tomarlo si cumple ambas restricciones. Ambas operaciones se resuelven en tiempo constante, ya que solo requieren acceder a celdas previamente calculadas de la tabla.

TRANSICIÓN Y ACTUALIZACIÓN DE ESTADOS

La transición mat[i][s][l] = max(no tomar, tomar) aprovecha el hecho de que la matriz se construye fila por fila, garantizando que los resultados necesarios para computar mat[i][s][l] ya fueron calculados en la fila anterior i − 1. La operación de tomar un archivo requiere un acceso directo a mat[i-1][...], por lo que no agrega complejidad extra. Así, cada estado se procesa de forma independiente sin recurrir a evaluaciones adicionales que incrementen la complejidad.

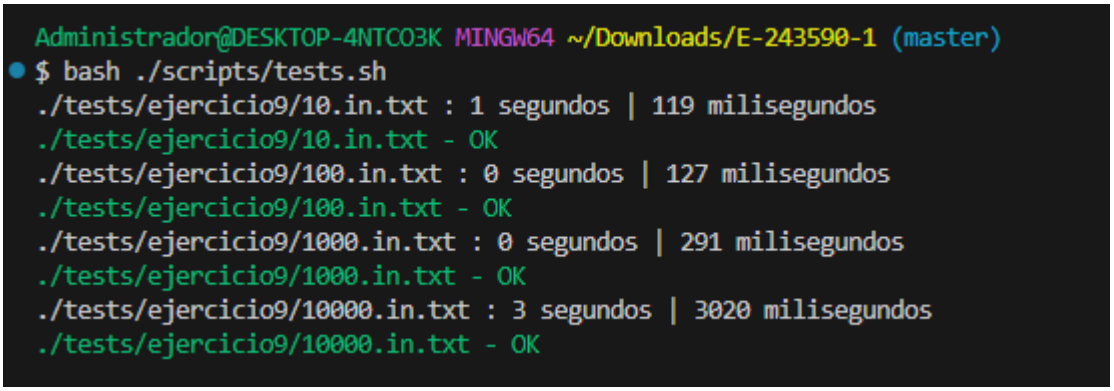
ALMACENAMIENTO Y ACCESO A RESULTADOS

La tabla tridimensional utiliza memoria proporcional a O(N·S·L), lo cual es necesario para almacenar todas las posibles combinaciones. Cada celda almacena un entero, y la operación de lectura y escritura es siempre O(1). Aunque el tamaño de la matriz podría ser

considerable, el diseño del ejercicio permite manejarla dentro de los límites de ejecución al tratarse de una DP iterativa y no recursiva.

La complejidad temporal completa de la solución es O(N·S·L), ya que se recorren las tres dimensiones de la tabla y cada transición cuesta tiempo constante.

Resultados De Prueba



Ejercicio 10

Para resolverlo se utilizó un algoritmo de búsqueda exhaustiva que recorre el mapa intentando encontrar la celda que contiene el carácter buscado. La solución toma cada centro de distribución por separado, ejecuta el algoritmo de backtracking definido en resolverFC y mide la cantidad de pasos utilizados para llegar al objetivo. Luego, entre todos los centros evaluados, se selecciona aquel que presenta la menor cantidad de pasos requeridos. Para realizar este ejercicio nos basamos en el ejercicio realizado en clase caballo.cpp.

Justificación de la complejidad temporal

RECORRIDO DEL MAPA MEDIANTE BACKTRACKING

Cada ejecución de resolverFC sobre un mapa de tamaño M×N recorre las celdas verificando movimientos válidos. Esto da una complejidad acotada por O(M·N).

EVALUACIÓN DE MÚLTIPLES CENTROS

Como el algoritmo se ejecuta para cada uno de los P centros de forma independiente, la complejidad total acumulada es O(P·M·N).

COMPARACIÓN Y SELECCIÓN

Elegir el mejor centro solo requiere comparaciones constantes, contribuyendo O(1) por instancia.

La complejidad final del ejercicio es O(P·M·N), ya que el costo dominante es ejecutar el backtracking en cada centro.

Resultados De Prueba

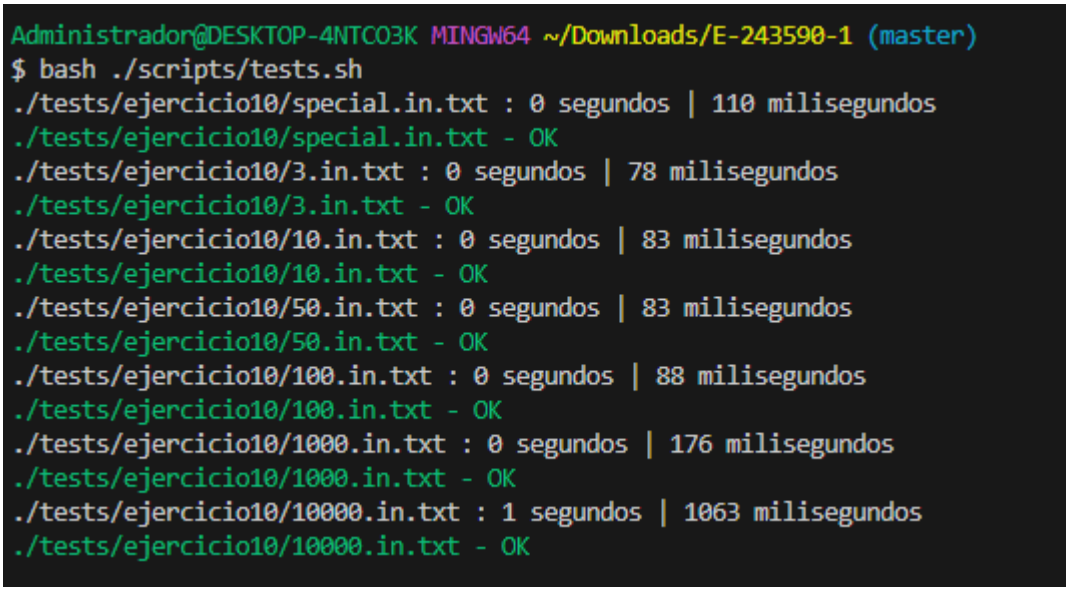


Tabla Resultados final

Ejercicio	Resultado
6	Completo
7	Completo
8	Completo
9	Completo
10	Completo