

ADVENT OF CODE 2024

Jose Daniel Martínez Mendoza y Javier Vidal Ochando

Introducción

La tarea asignada era la de completar varios problemas del Advent of Code 2024 utilizando técnicas y estructuras de datos vistas en la asignatura de Programación Avanzada. Para ello, se han resuelto varios problemas cuyas soluciones se encuentran en el siguiente [repositorio](#) de GitHub.

Se han resuelto 4 problemas:

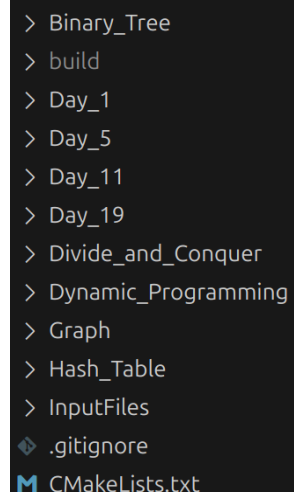
- Día 1: divide y vencerás.
- Día 5: grafos.
- Día 11: programación dinámica y árboles binarios
- Día 19: tablas hash

Se han agrupado dos temas de la asignatura en el día 11 debido a que no hemos sido capaces de encontrar un problema que se pudiese resolver dedicado únicamente a árboles binarios. Más adelante se explicará porque se ha decidido implementarlos justo en ese problema en concreto.

El repositorio está dividido por carpetas, cada día tiene su carpeta donde se puede encontrar el archivo principal donde se resuelve el problema y el input con el que se ha resuelto.

Las clases y funciones relacionadas con cada tema de la asignatura también han sido agrupadas por carpetas para poder encontrar con mayor claridad el código que tiene que ver directamente con la asignatura.

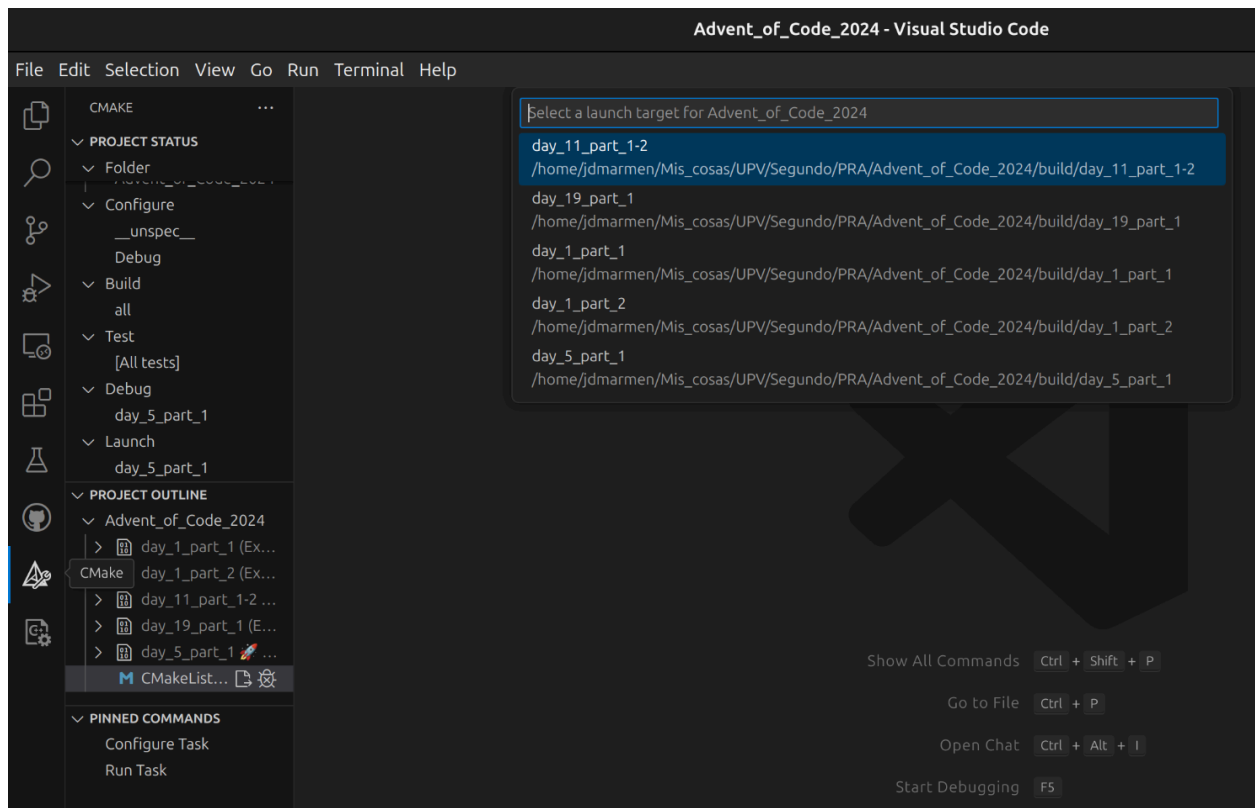
Además, se ha intentado hacer uso de la programación orientada a objetos en la medida de lo posible implementando bastantes métodos con funciones estáticas para que el archivo principal donde se resuelve el problema no esté tan sobrecargado y sea más legible. Por último, hay una clase InputFiles para manejar la apertura y el cierre de ficheros de manera más automática y limpia. En ella, también se han implementado algunos métodos para facilitar el almacenamiento de los datos que se incluyen en el fichero, aunque no se han usado en la resolución de todos los problemas.



```
> Binary_Tree
> build
> Day_1
> Day_5
> Day_11
> Day_19
> Divide_and_Conquer
> Dynamic_Programming
> Graph
> Hash_Table
> InputFiles
◆ .gitignore
M CMakeLists.txt
```

Por último, dado que hay varios archivos distintos y los métodos relacionados con los temas de la asignatura están también divididos en archivos de cabecera y archivos de implementación, para facilitar la compilación y ejecución del código se ha optado por hacer uso de CMake. Si a la hora de corregir las soluciones se desea probar algún caso en concreto o simplemente resulta más sencillo revisar el código desde un IDE, es tan sencillo como clonarse el repositorio, instalar la extensión de CMake en VSCode, configurar la carpeta donde se encuentra el código e ir al

apartado donde pone “Launch”, al pulsar sobre el lápiz saldrá un desplegable para elegir el código del día que se desea ejecutar, una vez seleccionado, es tan fácil como darle al botón de Play a la derecha de “Launch” y el código se compilará y ejecutará sin necesidad de introducir ningún comando por la terminal. Si se desea probar con otros puzzles, únicamente hay que cambiar el contenido de los archivos day_n_puzzle.



Día 1: Divide y vencerás

Se ha implementado el algoritmo de "Divide y Vencerás" utilizando el método de Merge Sort para resolver el problema. Merge Sort divide el vector en mitades, ordena cada mitad recursivamente y luego las combina en un solo vector ordenado. Este enfoque es eficiente con una complejidad de $O(n \log n)$.

Otras alternativas descartadas incluyen:

- Quick Sort: Aunque también es $O(n \log n)$ en promedio, su peor caso es $O(n^2)$.
- Bubble Sort: Tiene una complejidad de $O(n^2)$, lo que lo hace ineficiente para grandes conjuntos de datos.

Valoración personal:

La técnica de "Divide y Vencerás" aporta una solución eficiente y escalable, asegurando un rendimiento consistente incluso con grandes volúmenes de datos. Dado que para este problema era necesario tener el vector ordenado, se optó por ordenarlo de una manera eficiente con uno de los algoritmos aprendidos en clase. Es evidente que para vectores pequeños y problemas simples como este hay una gran diferencia a la hora de ordenar el vector con un algoritmo o con otro, pero al tratar un volumen mayor de datos o programar sistemas embebidos con recursos más limitados donde cada operación cuenta, es interesante utilizar técnicas eficientes y rápidas para la resolución de problemas.

Día 5: Grafos

Se ha implementado el uso de grafos para resolver el problema del día 5. Se ha creado una clase Graph que permite representar el grafo mediante listas de adyacencia. El grafo se construye leyendo aristas desde un archivo y añadiéndolas a la estructura. Luego, se verifica si las líneas de números están ordenadas según las conexiones del grafo.

Otras alternativas descartadas incluyen:

- Matrices de adyacencia: Aunque permiten una verificación rápida de conexiones, consumen más memoria, especialmente para grafos dispersos.
- Listas de incidencia: Son útiles para ciertos tipos de problemas, pero no son tan intuitivas para verificar el orden de los nodos.

Valoración personal:

El uso de grafos aporta una solución clara y estructurada para verificar el orden de los elementos según las conexiones definidas. Esta técnica es eficiente en términos de memoria y permite una fácil expansión del problema a grafos más grandes y complejos. Dado que el problema requería verificar el orden de los elementos basándonos en sus

conexiones, se optó por utilizar grafos, una de las estructuras de datos aprendidas en clase. Es evidente que para problemas más complejos y con mayores volúmenes de datos, el uso de grafos proporciona una solución eficiente y escalable, asegurando un rendimiento consistente y una fácil gestión de las relaciones entre los elementos.

Día 11: Programación dinámica y Árboles binarios

Se ha implementado el uso de Programación Dinámica y Árboles Binarios para resolver el problema del día 11. La clase `Dynamic_Programming` utiliza un enfoque de memorización para optimizar el cálculo de combinaciones de piedras y pasos. Se emplea un diccionario basado en un árbol binario de búsqueda (`BSTreeDict`) para almacenar y recuperar resultados previamente calculados, evitando así cálculos redundantes.

Programación Dinámica:

La función `count` de la clase `Dynamic_Programming` calcula el número de combinaciones posibles de piedras y pasos. Utiliza memorización para almacenar resultados intermedios en un diccionario (`memoDict`), lo que mejora significativamente la eficiencia al evitar recalcular los mismos valores múltiples veces. Esto se ha notado al completar la parte 2 del problema, ya que los números crecen rápidamente al realizar parpadeos y pese a que el código funcionaba perfectamente sin programación dinámica para 25 parpadeos, para 75 hubiese tardado demasiado. Además, para evitar problemas de overflow a causa del tamaño de los números, se ha optado por usar `uint64_t` en vez de `int`.

Árboles Binarios:

El diccionario de memorización se implementa utilizando un árbol binario de búsqueda (`BSTreeDict`). Este enfoque permite una inserción y búsqueda eficientes de los resultados intermedios, con una complejidad promedio de $O(\log n)$ para ambas operaciones. Debido a que no hemos sido capaces de encontrar un problema completamente dedicado a árboles, hemos pensado que era buena idea incluirlos en este, ya que al imaginar visualmente la estructura de las piedras originales y como se van creando nuevas piedras, nos dimos cuenta de que, ya que una sola piedra solamente puede partirse en 2 nuevas, cumplía perfectamente la estructura de árbol binario.

Otras alternativas descartadas incluyen:

- Tablas hash: Aunque ofrecen una búsqueda y almacenamiento rápidos, pueden tener problemas de colisiones y no garantizan un rendimiento $O(\log n)$ en el peor caso.
- Vectores o matrices: No son adecuados para este problema debido a la necesidad de manejar claves complejas y dinámicas.
- No utilizar memorización y esperar el tiempo que haga falta a que se ejecute el código para encontrar la solución al problema, desde luego no era una buena opción.

Valoración personal:

El uso combinado de Programación Dinámica y Árboles Binarios proporciona una solución eficiente y escalable. La memorización reduce drásticamente el tiempo de ejecución al evitar cálculos redundantes, mientras que el árbol binario de búsqueda garantiza un acceso rápido a los resultados almacenados. Esta técnica es especialmente útil para problemas complejos con múltiples subproblemas superpuestos, como el del día 11. La implementación demuestra cómo combinar diferentes estructuras de datos y técnicas algorítmicas puede llevar a soluciones óptimas y eficientes, asegurando un rendimiento consistente incluso con grandes volúmenes de datos. Además, como hemos comentado antes, el árbol binario describe perfectamente la estructura de las piedras y sus particiones, por lo que encajaba perfectamente en este problema y haber visto este tipo de estructuras en clase ha ayudado a tener un enfoque mucho más visual para resolver el problema.

Día 19: Tablas Hash

Se ha implementado el uso de Tablas Hash para resolver el problema del día 19. La clase `HashTable` permite almacenar y buscar patrones de manera eficiente. Se utiliza una tabla hash para verificar si una línea puede ser construida a partir de los patrones disponibles.

Tablas Hash:

La clase `HashTable` implementa una tabla hash con encadenamiento para manejar colisiones, de manera que si se inserta más de un elemento que después de pasar por la función hash devuelve la misma clave, se inserta en una lista enlazada con todos los elementos que tienen esa clave. La función `hashFunction` calcula el índice de la cubeta para una clave dada, y las funciones `insertItem` y `searchItem` permiten insertar y buscar claves en la tabla, respectivamente.

Otras alternativas descartadas incluyen:

- Árboles binarios de búsqueda: Aunque permiten una búsqueda ordenada, tienen una complejidad de $O(\log n)$ en promedio, que es mayor que la búsqueda en tablas hash en el mejor caso.
- Listas enlazadas: No son adecuadas para búsquedas rápidas, ya que tienen una complejidad de $O(n)$ para la búsqueda.
- Gestión de las colisiones mediante doble hashing o sondeos lineales o cuadráticos: son otras formas vistas en clase de resolver colisiones que hubieran sido completamente válidas, pero hemos visto más sencillo y rápido el usar una lista enlazada a diseñar un nuevo método hash, ya que este, debería haber sido muy eficiente para evitar que se formasen nuevas colisiones.

Valoración personal:

El uso de Tablas Hash proporciona una solución eficiente para el problema de verificación de patrones. La inserción y búsqueda en la tabla hash tienen una complejidad promedio de $O(1)$, lo que permite manejar grandes volúmenes de datos de manera rápida. Esta técnica es especialmente útil para problemas que requieren búsquedas frecuentes y rápidas, como el del día 19. La implementación demuestra cómo las tablas hash pueden ser utilizadas

para almacenar y buscar patrones de manera eficiente, asegurando un rendimiento consistente incluso con grandes conjuntos de datos. Es evidente que el unico problema que presentan es el de las colisiones, pero gracias al contenido visto en clase, hemos sido capaces de gestionarlas.