**Christian Mauricio Zuluaga Mora**

**SOLUCIÓN BACK-END - DEVELOPER TECHNICAL SKILLS**

1. **CODING CHALLENGE**

*En el repositorio se encuentran dos soluciones al problema, una solución en C++ orientada al concepto de programación competitiva que fue subida a hackerRank y obtuvo todos los casos de prueba correctos. Y una solución web donde me enfoco más en los patrones de diseño y arquitectura de software y que igualmente funciona con los limites que tiene la descripción del problema.*

*Puede encontrar la solución web en el siguiente enlace* [*https://chzuluaga.github.io/Rappi-Technical-Test/index.html*](https://chzuluaga.github.io/Rappi-Technical-Test/index.html)

**Explicación algorítmica de la solución:**

La complejidad algorítmica de la solución es O(M2)

Ya que el numero de preguntas a lo mas es 1000 podemos almacenar todas las coordenadas dadas en las preguntas junto con su valor y cada vez que haya una consulta de tipo *QUERY* recorremos el arreglo de esas coordenadas y sumamos el valor de la casilla de aquellas que se encuentren entre las dos coordenadas de la pregunta.

Así nos evitaremos declarar y recorrer toda la matriz completa para cada pregunta ya que recorreremos a lo más 1000 coordenadas en el peor caso.

**Aplicación web:**

Para el Coding Challenge decidí usar un framework que me facilitara la creación de una aplicación web además de brindar flexibilidad al momento de aplicar el patrón de arquitectura **Modelo-Vista-Controlador**.

Angular me permite dividir la interfaz grafica, de la lógica de negocio y el modelo de datos, además que ofrece la creación de los llamados *services* para usar la inyección de dependencias.

El componente principal de la aplicación es *HOME* y posee tres tipos de archivos

.Html, .scss , .ts, spec.ts,

Los dos primeros se encargan de la creación y administración de la interfaz de usuario, el *.ts* es el encargado de la lógica de negocio y *spec.ts* es usa para test unitarios

**Capa de presentación:** Archivos *.html* se encargan de la recolección y presentación de la información desde y hacia el usuario. [Home.component.html](http://Home.component.html)

**Capa de lógica de negocios:** El controlador [home.component.ts](http://home.component.ts) se encarga de enlazar el modelo de datos y la vista para resolver el problema. Se apoya en dos *Services*, el primero input.service encargado de la validación de la entrada respecto a los requisitos que exige la descripción del problema, el segundo servicio es matrix.service encargado de la parte algorítmica que resuelve el problema

* 1. **Domain model:** Son todos los archivos creados en la carpeta shared, define cada una de las clases que se asocian para dar solución al problema. Permite fácilmente aplicar el patrón de diseño de responsabilidad única.

**Responsabilidad de las Clases:**

* + 1. **MatrixService:** Encargada del estado de la matriz (actualización y consulta de casillas)
    2. **InputService:** Encargada de la validación y consistencia de la entrada de prueba
    3. **Input:** Representación de las características de una entrada de prueba
    4. **Case:** Representación de las características de una caso de prueba
    5. **Matrix:** Representación de las características de la matriz
    6. **Coordinate:** Representación de las características de una coordenada
    7. **Query:** Representación de las características de una Pregunta
    8. **Constants:** Encargada de las constantes usadas durante el código

1. **CODE REFACTORING**
   1. Malas prácticas de programación evidenciadas en el código y como se superaron
      1. **Tener código *commented-out*** es decir código que comentamos para que no sea ejecutado. Esta clase de código debe ser eliminado y de ser necesario sería fácilmente recuperable usando un sistema de control de versiones.

**Solución:** Eliminar el código comentado.

* + 1. **Codificar en español**: Nombres de variables, funciones o comentarios en español es una mala práctica de programación ya que la mayoría de los lenguajes son creados en inglés y nuestro código luciría como una combinación de idiomas, además de que algunos compiladores no soportan la presencia de caracteres especiales como por ejemplo la *‘ñ’***,** y esta clase de errores son muy difíciles de depurar.

**Solución:** Cambiar variable servicio a service y crear todos los comentarios en ingles.

* + 1. **Comentarios poco claros** como por ejemplo //up carro no presentan ninguna información al desarrollador y se encuentran fuera de contexto.

**Solución:** Eliminar comentarios fuera de contexto.

* + 1. **Mezclar comillas simples con dobles:** Aunque en muchos lenguajes no representa ningún error, es una mala practica ya que debemos unificar la forma de codificar para que revisiones futuras sean fácilmente legibles.

**Solución:** Usar solo comillas simples.

* + 1. **Función muy extensa:** La función es demasiado extensa y puede traer consigo problemas de legibilidad. Podemos dividir su funcionalidad en varios métodos, lo que aumentaría la calidad y claridad del código.

**Solución:** Dividir la funcionalidad en distintas funciones con la creación de nuevas clases. La parte relacionada a la actualización del servicio la maneja la función *update\_service* de la clase *Service* y la parte de notificación la maneja la función *notify\_user* de la clase *Notifier*. Con esto obtenemos un código mas limpio y legible.

* + 1. **NO uso de constantes:** En el código NO se usan constantes para muchos valores y en el caso de que haya que hacer un cambio, aquel cambio debe hacerse en cada una de las líneas de código donde usábamos esos valores. Es más fácil crear constantes y cada vez que sea necesario se le cambia solo el valor a la constante.

**Solución:** Creación de la clase *Constant* donde almacenaremos todas las constantes de nuestro código, de ser necesario cambiar un valor solo tendrá que hacerse en esta clase y no en todo el código.

* + 1. **No hay comentarios que describan la funcionalidad del código:** Agregar un pequeño comentario antes de la declaración de una función y durante el código es una buena practica de programación ya que permite en revisiones futuras entender más fácil lo que programamos.

**Solución:** Agregar comentarios que describan las funciones.

* + 1. **Existencia de instrucciones duplicadas** que podrían ser almacenadas en una variable como el caso de *Input::get(‘driver\_id’))*

**Solución:** Creación de la variable *$driver\_id* y omisión de la reconsulta de la variable *$service.*

1. **PREGUNTAS**
   1. ¿En qué consiste el principio de responsabilidad única? ¿Cuál es su propósito?
      1. El principio de responsabilidad única establece que un clase debe tener un único propósito o una sola razón para cambiar. El objetivo de este principio es minimizar el impacto negativo de los cambios sobre el código para favorecer la mantenibilidad de un proyecto, se sugiere identificar las responsabilidades y sepáralas en distintas clases o módulos que se encarguen de una sola funcionalidad y que se relaciones entre sí.
   2. ¿Qué características tiene según su opinión “buen” código o código limpio?
      1. Debe aplicar SOLID (5 principios fundamentales de OOP)
         1. Principio de responsabilidad única
         2. Abierto para su extensión, pero cerrado para su modificación
         3. Una clase derivada se comporta como su clase base
         4. Interfaces simples con pocos métodos
         5. Clases de alto nivel no deben depender de clases de bajo nivel
      2. Debe ser legible y conciso, de fácil entendimiento para futuros desarrolladores.
      3. No debe ser redundante (código duplicado).
      4. Debe poseer pruebas unitarias y de aceptación.
   3. ¿Qué es un microservicio?, ventajas y desventajas de los microservicios
      1. Es una pequeña división de la funcionalidad de un software en componentes (microservicio) que se comunican entre ellos con la característica de que se pueden desplegar de forma independiente.

Su principal ventaja es la facilidad al cambio y la escalabilidad del software, y así mismo como desventaja posee la gran complejidad de administración y gestión de un gran numero servicios además de que los llamados web son más costosos que lo llamados internos que ejecuta una aplicación en el enfoque monolítico