**INFORME DESAFIO 2**

Juan José Quiceno Pabón

Programa de Ingeniería en Telecomunicaciones

Universidad de Antioquia

Facultad de Ingeniería

Augusto Salazar

Aníbal Guerra

Medellín, Colombia

2024

**ANÁLISIS INICIAL**

En las primeras observaciones respecto al desafío planteado, puedo identificar un problema que no es particularmente complejo en términos de lógica aplicada. Sin embargo, la verdadera dificultad radica en la elección de los algoritmos que implementaré, ya que, a diferencia del primer desafío propuesto, creo que el tiempo de codificación se reducirá significativamente, pero vendrá acompañado de una mayor inversión de tiempo en el análisis y diseño de la solución.

En resumen, se solicita desarrollar una red metro, la cual tendrá diferentes líneas interrelacionadas por medio de estaciones de transferencia en cada una. Estas líneas tendrán n números de estaciones al gusto del usuario con un x tiempo de recorrido entre cada una de ellas, para que así, a la hora de simularlo se puedan hacer los respectivos cálculos teniendo en cuenta la eliminación o el añadido de algunas estaciones durante la ejecución del programa.

Como ideas iniciales me surge una estructura de datos muy sencilla para una red metro con una sola línea, un arreglo con n espacios donde se almacenaran cada una de las estaciones, pero en el caso de una red con múltiples líneas la metodología cambia, ya que ahora tengo que agregar un arreglo extra que contenga esas nuevas estaciones, y la idea es que en el caso de una nueva línea, en el espacio de la estación de transferencia se guarde ese arreglo, de esta forma no hay que manejar arreglos independientes y desconectados del resto, teniendo siempre como referencia la primera estación que representara la transferencia entre líneas. Ya en cuanto al tiempo de recorrido entre estaciones, tengo varias ideas, pero no muy claras aún, por lo que dejare eso a una etapa posterior del desarrollo.

Ya para la parte de simulación, en vista de que el cálculo del recorrido solo puede hacerse en 1 línea especifica, simplifica más las cosas, ya que es aplicar una suma entre los elementos que acompañan las estaciones dentro de los arreglos.

**Otras posibles implementaciones:**

* Darle a la estructura de datos una forma bidimensional, donde haya n arreglos que almacenen cada una de las estaciones junto con su tiempo para llegar a la siguiente estación.
* En el caso de otra línea es lo mismo, solo que la primera estación será la referenciada como estación de transferencia.

**ETAPAS DEL DESARROLLO**

1. **Etapa Inicial del Desarrollo**: Durante esta fase inicial, nos enfocaremos exclusivamente en el análisis del problema y en la construcción del informe basado en las conclusiones obtenidas. La creación del algoritmo consistirá en la inclusión de ideas claras y concisas que nos ayudarán a definir el siguiente paso en el proceso de desarrollo. Cabe mencionar que el progreso se reflejará en el repositorio “Desafio\_2” en GitHub a medida que avancemos día a día.
2. **Etapa de Codificación**: En esta etapa, daremos prioridad a establecer las bases del programa. Comenzaremos por implementar los métodos y funciones más sencillos, para luego ir construyendo progresivamente el algoritmo final. Evitaremos abordar directamente los métodos o funciones más complejas del programa, como, por ejemplo, el cálculo del tiempo de recorrido entre estaciones, sin antes tener una base de datos bien estructurada.
3. **Etapa de Pruebas**: Durante este período, supervisaremos el comportamiento del programa durante su ejecución y las diferentes respuestas que proporciona ante diversos escenarios, como excepciones o datos de entrada incorrectos. Esta supervisión permitirá reforzar aún más el algoritmo, gracias a los problemas que surjan durante la ejecución.

* Todas estas etapas se verán reflejadas en los avances detallados al final del informe, que sirven como bitácora al proceso realizado.

**¿CÓMO FUNCIONA EL PROGRAMA?**

El desarrollo a la solución esta dividido en 2 partes, un segmento de código dedicado a la manipulación de las estructuras de datos que conforman a la red metro, y otro segmento que se encarga de mostrar información general o especifica de esta, todo con el fin de encapsular los diferentes requerimientos del problema y que el desarrollo fuera más sencillo.

Todo el proceso de análisis y construcción del código fuente desemboca dentro de la clase **Metro,** esta, como bien dice su nombre, intenta simular como seria el funcionamiento de una red metro mediante la administración de componentes como líneas y estaciones que modifican el tiempo de recorrido entre ellas y en una clase interna de **Metro** llamada **Simulation** que toma los elementos de **Metro** por medio de un punterocon el fin de simular un objeto “pasajero”, el cual trata de representar el viaje de una persona dentro de la red. Estas clases se componen de:

* **ATRIBUTOS**:
  + **stations**: Es la estructura de datos principal, la cual es un arreglo de arreglos de tipo cadena de caracteres, en la que se encuentran las diferentes estaciones divididas en segmentos que representan sus líneas.
  + **hour**: Es la estructura de datos (tipo entero) que se encarga de almacenar los tiempos de recorrido entre estaciones, esta fue diseñada intencionalmente para que este alineada con la estructura principal en base a ciertas proporciones que dictan los patrones encontrados durante el proceso de análisis. Esta, al igual que la estructura madre está dividida en segmentos que representan su línea en la red.
  + **total\_lines**: Como su nombre ya lo describe, es un atributo de tipo entero el cual se encarga de almacenar la cantidad líneas totales que se encuentran actualmente en la red, esto con el fin de darle a numerosos ciclos dentro del código un limite en sus iteraciones o también servir de índice en algunos arreglos.
  + **act\_line**: Este atributo de tipo entero representa la línea actual sobre la que se esta trabajando durante la ejecución.
  + **len**: Atributo de tipo entero que representa el tamaño de una línea sobre la cual el flujo del programa está trabajando.
* **MÉTODOS**:
  + **Metro**: Este es el constructor de la clase, la razón por la que no recibe ningún parámetro es que inicializa la estructura de datos con una línea de metro con sus respectivas características por defecto, ya si el usuario decide eliminarlo, lo hará durante el transcurso de la ejecución.
  + **Admin e Info**: La razón por las que están juntas es porque su funcionamiento es similar, le brindan al usuario un menú con el cual pueden dirigir la ejecución del programa a su gusto, en **Admin** manipulando las estructuras de datos y en **Info** consultando sobre ellas.
  + **ShowMe**: Esta encargada de graficar el estado de la línea sobre la cual se esté trabajando (determinado por **act\_line**), o también puede funcionar como menú al momento de elegir alguna estación.
  + **Restart**: Su complejidad a la hora de crearla no fue mucha, ni siquiera cuenta con muchas líneas de código, pero su funcionamiento es primordial en muchos segmentos del programa, este método lo que hace es cambiar los atributos dependiendo la línea que el usuario o el programador (en su versión sobrecargada) vaya a trabajar, ya que las características de la línea 1, por ejemplo, no son las mismas que las de la línea 3 que puede tener un numero diferente de estaciones.
  + **NewStation y RemStation**: Se encargan de añadir o eliminar una estación a la línea en la red que desee el usuario en base a ciertas restricciones.
  + **AddLines y RemLines**: Se encargan de añadir o eliminar líneas a la red metro en base a ciertas restricciones.
  + **Transf**: Le da la opción al usuario de escoger cual va a ser la estación que servirá como transferencia a la nueva línea.
  + **ConfHour**: Cuando se agrega una nueva línea, la estructura que contiene los tiempos de recorrido debe crecer de igual manera, lo que hace este método es agregar un nuevo espacio al arreglo.
  + **Time**: Cuando se agrega o elimina una estación, también se debe realizar el mismo proceso con los tiempos de la estación previa y la que sigue dependiendo el caso, por lo que este método precisa de 2 argumentos que son type (determina si se borra o elimina tiempos) y ubi (La ubicación de la estación dentro de la linea).
  + **Travel (Simulation)**: Este método perteneciente a la clase **Simulation** se encarga de calcular el tiempo de recorrido entre una estación y otra gracias a los atributos obtenidos de **Metro** a través de una instancia creada en la clase **Simulation**.
* Por otra parte, tenemos las siguientes funciones externas:
* **Validation (**Función amiga de **Metro)**: Recibe 3 enteros como parámetro, uno que corresponde al valor por evaluar y los otros 2 que corresponden al valor mínimo y máximo que puede tomar el primer parámetro, todo esto convertido a char para evitar datos de entrada que no pertenezcan al tipo. Resumiendo, hace la validación respectiva a todos los datos que el usuario ingrese.
* **Titan**: Cumple la función de enseñar al usuario el menú principal, el cual le da 4 opciones: Administrar red metro (Le permite manipular los elementos de la red), Consultar información (Consultas acerca de los elementos de la red), Simulación de recorrido (Simula el recorrido de una estación a otra) y salir del programa, dependiendo la elección invoca ciertos métodos por medio del objeto de tipo **Metro** creado.

**AVANCES**

* **Día 1 – 25 de abril**:
* Se comienza con el proceso de análisis del problema y desarrollo del informe.
* **Día 2** – **26 de abril:**
* Se define la metodología y el orden que se seguirá durante el desarrollo del problema en el informe.
* **Día 3** – **26 de abril:**
* Se creo una clase **Metro**, que de momento será más que suficiente para el desarrollo del programa.
* Se realizan diferentes pruebas para determinar cuál será la mejor forma de administrar una base de datos con 1 o varias líneas de metro.
* Se realizaron pruebas en Online GDB con un arreglo unidimensional que almacena 5 estaciones, allí se colocaron a prueba 2 métodos, uno que agregaba una nueva estación y otro que las eliminaba.
* **PROBLEMAS**: Al realizar pruebas con arreglos unidimensionales, no imaginé como aplicar esos algoritmos a un arreglo que contenga varias líneas, ni de como modificar el arreglo principal para agregar esa línea.
* Surge una idea para manejar el arreglo principal, esta consta en crear sub-arreglos en cada método que maneje la línea que se especifique.
* Surge una idea de tener un método que me permita restablecer distintas variables en base a la línea de metro que se desee modificar.
* **Día 4** – **28 de abril:**
* Se comenzó a codificar con las ideas que surgieron durante los últimos días, resultando en un gran avance en la parte de manejo de la estructura de datos principal.
* Se definieron los atributos que conforman la clase **Metro**.
* Se decidió que el arreglo principal será tratado como un arreglo de arreglos, siendo fiel al desafío propuesto con una red que contiene diferentes líneas.
* Se logro desarrollar diferentes métodos que permiten añadir estaciones y líneas, consultar la información general del sistema y de líneas en específico.
* Para llevar un orden durante la ejecución, se aplicaron diferentes menús mediante switch para manejar el flujo del programa a gusto del usuario.
* **PROBLEMA**: Todavía no se tiene muy claro cómo se alinearán los datos correspondientes al tiempo con la estructura de datos diseñada.
* **Día 5** – **29 de abril:**
* Se corrigieron múltiples algoritmos para la parte de las estaciones, entre ellos los que añaden / eliminan estaciones y el que elimina líneas, además de que se crearon otros menús que le permiten al usuario un uso más cómodo del programa.
* **Día 6** – **30 de abril:**
* Se realizo un gran avance a través del día 30 de abril, en este se afianzo la idea de la estructura de datos principal, la cual estaría conectada a través de estaciones de transferencia, que serán la primera estación dentro de cada sub-arreglo, acompañado de diferentes validaciones de entrada que evitan eliminar estas estaciones de transferencia.
* Se determino que solo se podrá eliminar la última línea de metro agregada, esto a consecuencia de las aclaraciones realizadas por el profesor en clase, donde dejo claro de que no se podrá eliminar conexiones o transferencias dentro de la red.
* Se realizaron varias correcciones en cuanto a la lógica dentro de los métodos que se encargan de manejar las estaciones, dejando así un muy buen desempeño durante unas pequeñas pruebas que se les realizo, donde cumplen sus labores justo como quiero.
* Se empezó de lleno con la estructura de datos ligada a los tiempos del recorrido entre estaciones, finalmente se decidió manejarla individualmente y separada del arreglo que contiene las estaciones para facilitar su uso alineándola.
* Se desarrollaron distintos métodos que manejaran el arreglo de tiempos en base a los algoritmos realizados para las estaciones.
* Se sobrecargó el método encargado de redirigir al usuario a distintas líneas de la red, para que ahora realice el mismo trabajo a gusto del programador.
* Se realizo un pequeño análisis para formar una idea de como será el método encargado de las validaciones de entrada.
* Se implemento un algoritmo que permite al usuario elegir, a través de un menú, la estación de transferencia de la línea creada, ya que antes podía crear una línea sin conectarla a una ya existente.
* Al no realizar los commits esperados hasta la fecha y en vista de que el avance fue significativo en un par de días, se subirán los respectivos cambios a 2 ramas aparte de la principal.
* **PROBLEMA**: Se encontraron múltiples problemas en cuanto a la lógica de los métodos que manejan el tiempo, principalmente en cómo se distribuye y/o crece la estructura de datos ante ciertos escenarios, por lo que se destinara una buena porción de tiempo en el análisis individual de este problema.
* **Día 7** – **1 de mayo:**
* Se cambio en un gran porcentaje los métodos correspondientes al tiempo de recorrido, añadiendo también uno que me ayuda con la parte de los tamaños dentro de los arreglos para después poder manejarlos al gusto.
* Se cambio la función de validaciones a ser una función amiga, y no un método como se había definido anteriormente.
* Se dedico un buen tiempo en el desarrollo del informe en base a los algoritmos agregados últimamente.
* **ERROR**: Se encontró un error en la función delete del método **Time**, lo que pasa es que cuando agrego 2 estaciones en la última posición de una línea de manera consecutiva este me produce un error, el cual, después de un largo análisis en búsqueda de su motivo, aun no he podido determinar.
* **Día 8 – 2 de mayo**:
* Con el fin de seguir algunas recomendaciones del profesor dadas en la sesión para aclarar dudas acerca del desafío 2, se decidió separar **Simulation** y a partir de ahora tratarla como una clase que hereda de **Metro**, ya que después de reflexionar un poco acerca del análisis previamente realizado. Se podría ver **Simulation** como una clase que representa el viaje de un pasajero a través de la red metro por medio del nuevo método **Travel**.
* Se dividieron los atributos en private y protected, para que así la clase que hereda solo tome lo que necesita.
* Horas después se decidió no utilizar el concepto de herencia para la clase **Simulation**, si no que ahora formara parte de **Metro**, obteniendo sus atributos por medio de punteros.
* Se corrigió el método **Time** por algunos índices mal colocados y se añadió la parte que elimina los tiempos cuando remuevo una estación de la línea con la misma lógica que su contraparte (la que añade), si se elimina una estación que esta entre otras 2, se elimina un espacio y se reemplaza el otro con la suma de esos 2.
* Se agrego más contenido al informe.
* **ERROR**: El ultimo espacio dentro de cada arreglo en **hour** esta ocupado por 0, solo para llevar siempre el mismo tamaño en cuanto a estaciones y sus tiempos, pero este también me sirve para las diferentes pruebas que le realizo, en ellas se puede evidenciar que a partir de cierto número de líneas el ultimo espacio no obtiene ese valor, a parte de que puede llegar a un error de segmentación si se agregan muchas líneas.