

## **Informe Proyecto 1**

### **Asignatura:**

Análisis y diseño de algoritmos

### **Programa:**

Ingeniería de Sistemas

### **Estudiantes:**

Joan Sebastián Saavedra Perafán - 202313025

Juan Manuel Hoyos Contreras - 202380796

Sebastián Cifuentes Florez - 202380764

### **Semestre:**

Tercer Semestre

### **Docente:**

Carlos Andres Delgado Saavedra

Universidad del Valle – Sede Tuluá

## LA TERMINAL INTELIGENTE

### Clase Transformación Dinámica

En esta clase se Implementa los procesos para calcular el costo mínimo de transformar una cadena de caracteres source en otra target utilizando diferentes operaciones.

Se hace una definición de las operaciones posibles con sus costos asignados que son los siguientes:

**Avanzar:** Moverse al siguiente carácter si coinciden (costo=1).

**Borrar:** Eliminar un carácter de la palabra source (costo=2).

**Reemplazar:** Sustituir un carácter en source con uno de target (costo=3).

**Insertar:** Agregar un carácter en source para que coincida con target (costo=2).

**Eliminar todo:** Borrar todos los caracteres restantes en source hasta el final (costo=1).

Inicializamos el nodo que representa un estado en el proceso de transformación, con esto se incluye la cadena actual de trabajo (source parcial), la operación realizada para llegar ahí y el costo acumulado.

En el cálculo del costo mínimo utilizamos programación dinámica para almacenar el costo mínimo de transformar los primeros  $i$  caracteres de source en los primeros  $j$  de target y cada celda de la tabla guarda el menor costo de realizar esa transformación. Al inicializar la raíz se empieza con la palabra source original y un costo de 0. Se comparan caracteres entre source y target para decidir qué operación realizar y se evalúan las opciones (avanzar, borrar, reemplazar, etc.) y se elige la de menor costo.

El método **calcularCostoMinimo** utiliza una matriz para calcular el costo más bajo posible de transformar una cadena source en otra target a través de diferentes operaciones permitidas. La matriz tiene dimensiones  $(m+1) \times (n+1)$  donde  $m$  es la longitud de source y  $n$  es la longitud de target. Creamos la matriz de costos donde  $\text{costos}[i][0]$  es el costo de convertir los primeros  $i$  caracteres de source en una cadena vacía y  $\text{costos}[0][j]$  es el costo de convertir una cadena vacía en los primeros  $j$  caracteres de target. La matriz se llena celda por celda, evaluando todas las operaciones posibles para transformar una porción de source en target. Para cada celda  $(i, j)$  se calculan los costos de realizar las operaciones y al finalizar, la celda  $\text{costos}[m][n]$  contendrá el costo mínimo total para transformar toda la cadena source en target.

El método **imprimirTablaDeCostos** imprime la matriz de costos de una manera organizada para visualizar cómo se calcularon los costos de transformar la cadena source en target.

## Terminal Inteligente

algorithm

altruistic

Matriz de Costos:

	a	l	t	r	u	i	s	t	i	c	
a	0	2	4	6	8	10	12	14	16	18	20
l	2	0	2	4	6	8	10	12	14	16	18
g	4	2	0	2	4	6	8	10	12	14	16
o	6	4	2	2	4	6	8	10	12	14	16
r	8	6	4	4	4	6	8	10	12	14	16
i	10	8	6	6	4	6	8	10	12	14	16
t	12	10	8	8	6	6	6	8	10	12	14
h	14	12	10	8	8	8	8	8	8	10	12
m	16	14	12	10	10	10	10	10	10	10	12
	18	16	14	12	12	12	12	12	12	12	12

### Clase transformación Voraz

Se transforma una cadena fuente en una objetivo mediante operaciones con costos predefinidos. En este enfoque se toman decisiones inmediatas para minimizar el costo en cada paso sin considerar todas las posibles combinaciones.

El método **transformar** evalúa las operaciones disponibles y selecciona la de menor costo en cada paso, acumulando el costo total y registrando las operaciones realizadas.

Las variables iniciales  $i$  y  $j$  son los índices para recorrer las cadenas fuente y objetivo, costo total acumula el costo de las operaciones realizadas y las operaciones son una lista que guarda una descripción de las operaciones realizadas para transformación. La lógica del while compara caracteres actuales donde  $\text{fuente}[i] == \text{objetivo}[j]$ , ambos caracteres coinciden realizando la operación advance, agrega el costo de avanzar y mueve los índices ( $i++$ ,  $j++$ ).

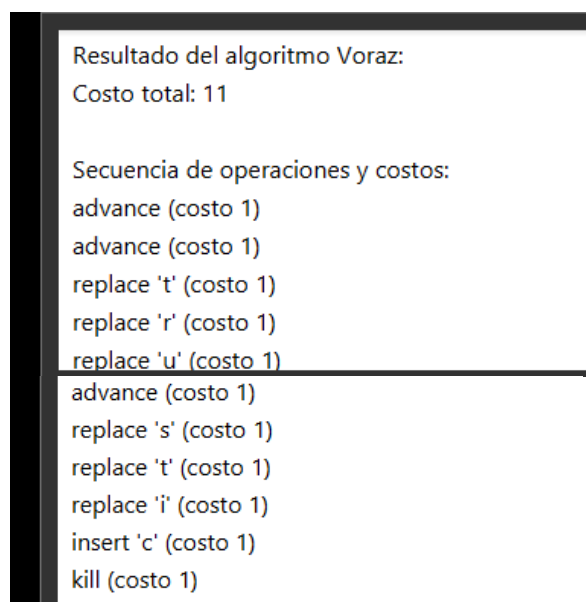
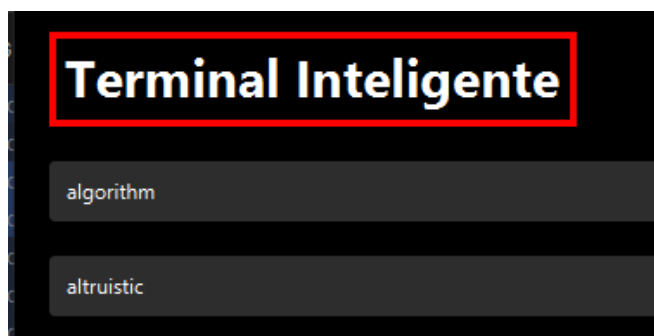
Si los caracteres no coinciden evalúa las operaciones posibles de reemplazar donde cambia fuente[ i ] por objetivo[ j ],

Insertar donde el carácter objetivo es [ j ] en fuente, borrar elimina fuente[i] y después se selecciona la operación más barata, compara los costos de las tres operaciones, al hacer este proceso realiza la operación de menor costo, ajusta la cadena fuente y actualiza índices (i, j) y el costo total.

En otro caso si quedan caracteres solo en fuente (i < fuente.length())y si fuente tiene caracteres sobrantes se realiza la operación kill donde se borran todos los caracteres restantes y agrega el costo correspondiente y termina el bucle.

Se crean unos métodos auxiliares y se encargan de realizar modificaciones en la cadena estado según las operaciones necesarias: eliminar, reemplazar o insertar caracteres. Estas utilizan la clase StringBuilder para facilitar las modificaciones, ya que las cadenas en Java son inmutables. Delete elimina un carácter en la posición especificada (índice) de la cadena estado, replace reemplaza el carácter en la posición especificada (índice) de la cadena estado por un nuevo carácter (letra), insert inserta un carácter (letra) en la posición especificada (índice) de la cadena estado, estos métodos son fundamentales para simular las operaciones de edición en la cadena fuente durante la transformación.

El método **imprimirOperaciones** imprime la lista de operaciones realizadas durante la transformación de una cadena y sus costos donde se puede visualizar los pasos.



## Fuerza bruta (solución ingenua)

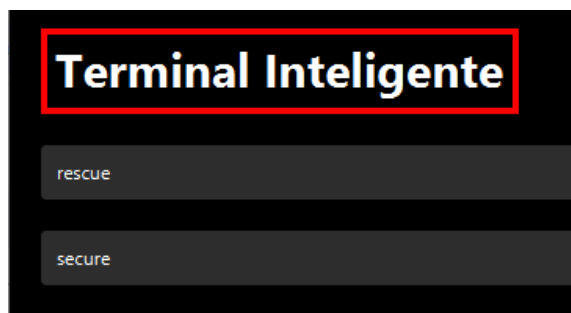
Se implementa un algoritmo de búsqueda en amplitud (BFS) para transformar una cadena inicial en una cadena objetivo mediante operaciones con el objetivo de encontrar la solución con el menor costo posible. Se inicia el árbol creando un nodo raíz con la cadena inicial, costo 0 y profundidad 0, se define el número de nodos repetidos para evitar expansiones innecesarias. Con el método **busquedaAmplitud** se usa una cola para explorar todos los nodos nivel por nivel, se expande cada nodo generando posibles hijos usando operaciones válidas, además de registrar el número total de nodos generados y los nodos repetidos y si encuentra un nodo que coincide con la cadena objetivo, lo guarda como solución.

En **generarHijos** se aplica las siguientes operaciones en el estado actual del nodo:

- Avanzar: Mueve el índice si los caracteres coinciden.
- Borrar: Elimina un carácter.
- Reemplazar: Cambia un carácter por otro del destino.
- Insertar: Agrega un carácter del destino.
- Kill: Borra todos los caracteres restantes de la cadena.

Se genera nodos hijos con la nueva cadena, costos y profundidades.

En la Impresión de la solución que es **imprimirRuta** se imprime el camino desde el nodo raíz hasta el nodo solución e incluye la operación realizada, el estado de la cadena y el costo acumulado.



	rescue	0
Insert	srescue	1
Insert	serescue	2
Insert	secrescue	3
Insert	securescue	4
Insert	securrescue	5
Insert	securerescue	6

```

Delete | securescue | 7
Delete | securescue | 8
Delete | securecue | 9
Delete | secureue | 10
Delete | securee | 11
Kill | secure | 12
Nodos generados en total: 107663
Nodos repetidos encontrados: 19477

```

Nodo que encontró la solución: Nodo{estado='secure', operador='Kill', profundidad=12, costo=12, padre=Nodo{estado='securee', operador='Delete', profundidad=11, costo=11, padre=Nodo{estado='secureue', operador='Delete', profundidad=10, costo=10, padre=Nodo{estado='securecue', operador='Delete', profundidad=9, costo=9, padre=Nodo{estado='securescue', operador='Delete', profundidad=8, costo=8, padre=Nodo{estado='secureescue', operador='Delete', profundidad=7, costo=7, padre=Nodo{estado='securerescue', operador='Insert', profundidad=6, costo=6, padre=Nodo{estado='securrescue', operador='Insert', profundidad=5, costo=5, padre=Nodo{estado='securscue', operador='Insert', profundidad=4, costo=4, padre=Nodo{estado='secrescue', operador='Insert', profundidad=3, costo=3, padre=Nodo{estado='serescue', operador='Insert', profundidad=2, costo=2, padre=Nodo{estado='srescue', operador='Insert', profundidad=1, costo=1, padre=Nodo{estado='rescue', operador='', profundidad=0, costo=0, padre=null}}}}}}}}}}}}

## Clase Nodo

La clase Nodo representa un elemento en el árbol de búsqueda utilizado para transformar una cadena inicial en una cadena objetivo. Cada instancia de esta clase encapsula información sobre el estado actual de la transformación y cómo se llegó a él. Los elementos claves son los siguientes

### Atributos:

- padre: Nodo padre que generó este nodo, usado para rastrear la ruta desde la raíz hasta este nodo.
- estado: La cadena actual que representa el estado del nodo.
- operador: La operación aplicada para llegar a este nodo (por ejemplo, "Insert", "Delete").

- profundidad: La profundidad del nodo en el árbol, indica cuántos pasos se han tomado desde la raíz.
- costo: El costo acumulado para alcanzar este nodo desde la raíz.
- índice: La posición actual en la cadena donde se está evaluando o aplicando operaciones.

#### **Constructor:**

- El nodo padre (padre).
- El estado actual (estado).
- La operación aplicada para llegar al estado actual (operador).
- La profundidad (profundidad).
- El costo acumulado hasta este nodo (costo).
- El índice actual (índice).

#### **Métodos Getters y Setters:**

- Proveen acceso y modificación a los atributos privados del nodo, como getEstado(), setEstado(), etc.

#### **Método toString:**

- Sobrescribe la representación del nodo como cadena para facilitar la depuración o impresión. Muestra:
- Estado actual.
- Operador aplicado.
- Profundidad en el árbol.
- Costo acumulado.
- Información del nodo padre.

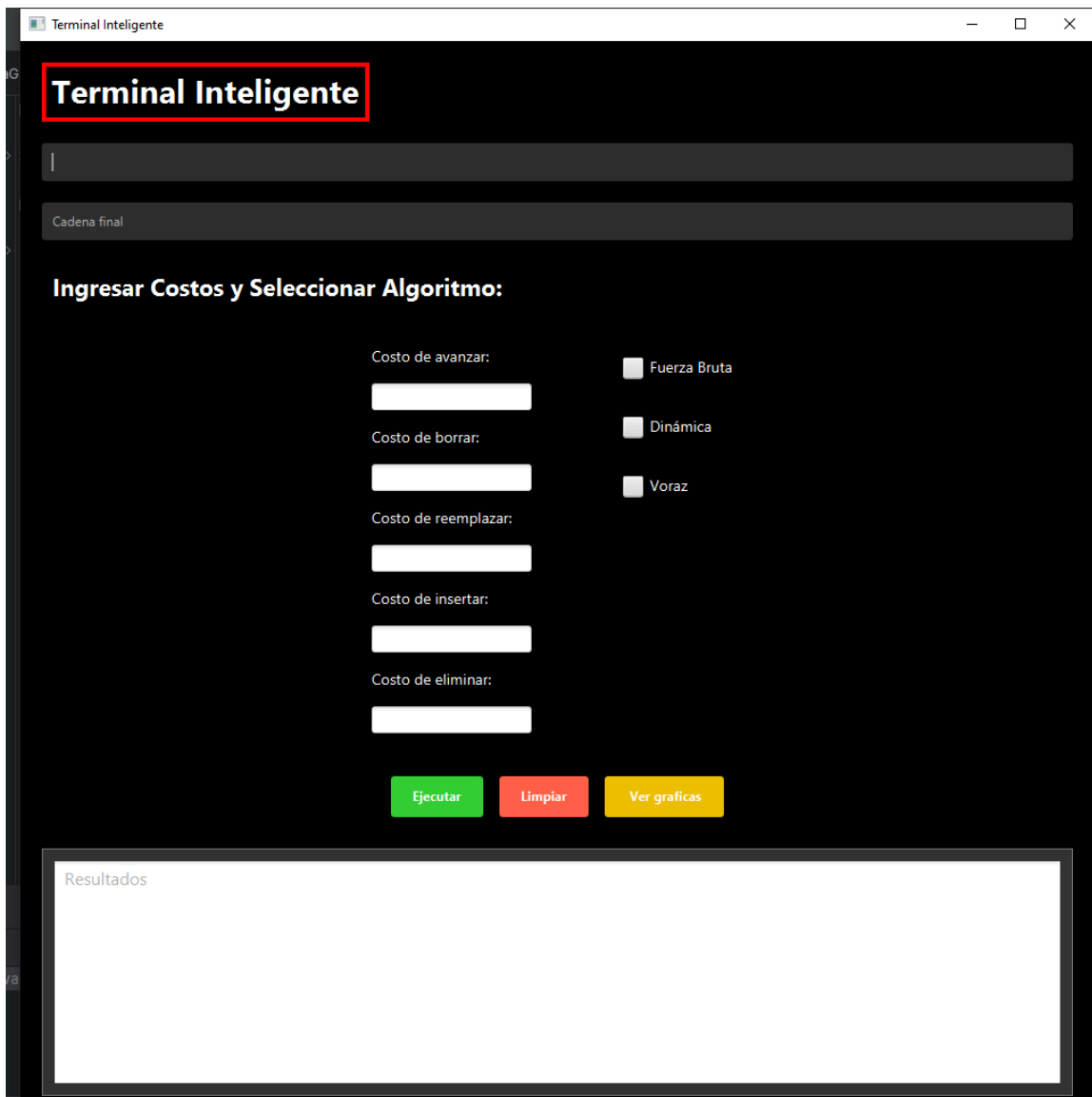
### **Clase Operations**

En esta clase se encapsula las operaciones posibles sobre un nodo para transformar su estado en otro. Estas operaciones serán esenciales para explorar posibles caminos en un árbol de búsqueda de transformaciones.

Los componentes que se encuentran es el nodo que representa el nodo actual sobre el cual se ejecutan las operaciones. Este nodo contiene el estado (cadena) y otros datos asociados, como su costo y profundidad. Operations(Nodo nodo) donde se inicializa la clase con un nodo específico, esto permite aplicar operaciones sobre el estado del nodo proporcionado. Cada

método realiza una operación específica sobre el estado del nodo. Todas las transformaciones usan un StringBuilder para modificar la cadena de forma eficiente. Avanzar simula avanzar al siguiente carácter en la cadena, devolviendo el nuevo índice, delete elimina el carácter en la posición índice de la cadena del nodo, replace(int indice, char letra) reemplaza el carácter en la posición índice por el carácter letra, insert(int indice, char letra) inserta el carácter letra en la posición índice de la cadena, kill(int indice) elimina todos los caracteres desde el índice hasta el final de la cadena. Esta clase se utiliza dentro del algoritmo de búsqueda para generar nodos hijos a partir de un nodo actual. Cada operación representa un cambio en la cadena del estado, lo que permite explorar nuevas transformaciones hacia el estado objetivo.

## Interfaz





Este código define la interfaz gráfica para trabajar con algoritmos relacionados con la transformación de cadenas. Cada sección está modularizada y estilizada cuidadosamente para mejorar la experiencia del usuario. Las operaciones principales se manejan mediante controladores de eventos asociados a los botones y campos de entrada.

### Componentes principales

- Título principal
- Campos de texto (TextFields)
- CheckBoxes
- Sección de costos
- Botones
- Área de resultados

## EL PROBLEMA DE LA SUBASTA PÚBLICA

### Clase voraz

Se tiene una entrada principal donde hay una lista de objetos, cada uno representando una opción de inversión con tres características, precio por acción (P) es cuánto se gana por cada acción de esa opción, mínimo de acciones (Mi) mínimo número de acciones que deben adquirirse para esta opción, máximo de acciones (Ma) es el máximo número de acciones disponibles para esta opción y total de acciones disponibles (A) el Número máximo de acciones que podemos comprar entre todas las opciones. `mutableList.sort(Comparator.comparingDouble(tripleta -> -tripleta.getP()))`; ordena las opciones en orden descendente por precio por acción (P), priorizando aquellas que generan mayor ganancia. El programa recorre la lista de tripletas para decidir cuántas acciones comprar de cada opción. Se actualizan ganancias y acciones restantes donde actualizar ganancias suma la ganancia de las acciones seleccionadas de la tripleta; el actualizar las acciones restantes resta las acciones compradas de las acciones disponibles. Se verifican condiciones de parada. La estrategia prioriza siempre las opciones más rentables y detiene la selección cuando no es posible cumplir las restricciones restantes.

```
maxAcciones: 50 accionesRestantes: 20050
maxAcciones: 130 accionesRestantes: 15040
7700
```

## Clase dinámica

La Estructura aborda  $p$  (precio) que es la ganancia por cada acción,  $mi$  (mínimo) es mínimo de acciones que se deben usar si es seleccionada y  $ma$  (máximo) el máximo de acciones que se pueden usar cada una de estas son asignadas a esta tripleta. La variable  $dp[j]$  representa la ganancia máxima alcanzable usando exactamente  $j$  acciones en total y esta tabla se inicializa en 0 con 0 acciones y la ganancia es 0. Con esto iteramos en orden decreciente desde el número total de acciones  $A$  hasta el mínimo requerido ( $mi$ ). El paso a seguir iteramos sobre las posibles cantidades de acciones ( $k$ ) que se pueden tomar de esta y solo consideramos valores dentro del rango  $[mi, ma]$  y verificamos que no excedemos el total disponible. El cálculo de la ganancia Máxima tomamos  $k$  acciones de la tripleta actual, la ganancia adicional es  $k * p$  y la ganancia acumulada previa es  $dp[j - k]$  que serán las acciones restantes después de usar  $k$ , actualizamos  $dp[j]$  con el valor máximo entre su estado actual y la nueva ganancia calculada.

```
operacion: 10 dp[4]: 0 dpj-k[3]: 0
operacion: 20 dp[4]: 10 dpj-k[2]: 0
operacion: 10 dp[3]: 0 dpj-k[2]: 0
operacion: 20 dp[3]: 10 dpj-k[1]: 0
operacion: 10 dp[2]: 0 dpj-k[1]: 0
operacion: 20 dp[2]: 10 dpj-k[0]: 0
operacion: 10 dp[1]: 0 dpj-k[0]: 0
operacion: 30 dp[4]: 20 dpj-k[2]: 20
operacion: 25 dp[4]: 30 dpj-k[1]: 10
operacion: 20 dp[3]: 20 dpj-k[1]: 10
operacion: 15 dp[3]: 20 dpj-k[0]: 0
operacion: 10 dp[2]: 20 dpj-k[0]: 0
operacion: 27 dp[4]: 30 dpj-k[3]: 20
operacion: 34 dp[4]: 30 dpj-k[2]: 20
operacion: 31 dp[4]: 34 dpj-k[1]: 10
operacion: 27 dp[3]: 20 dpj-k[2]: 20
operacion: 24 dp[3]: 27 dpj-k[1]: 10
operacion: 21 dp[3]: 27 dpj-k[0]: 0
operacion: 17 dp[2]: 20 dpj-k[1]: 10
operacion: 14 dp[2]: 20 dpj-k[0]: 0
operacion: 7 dp[1]: 10 dpj-k[0]: 0
34
```

## Fuerza bruta (solución ingenua)

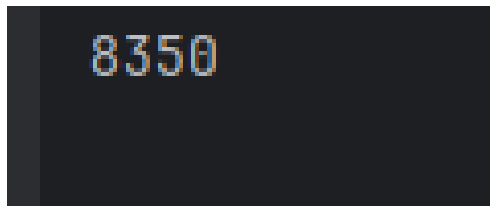
Hay un número total de acciones disponibles ( $A$ ), con esto se maximiza la ganancia utilizando tripletas,  $p$  (precio) es la ganancia por cada acción de esa tripleta,  $mi$  (mínimo) es

el número mínimo de acciones que se pueden tomar y  $m_a$  (máximo) será el número máximo de acciones.

Cada tripleta define el precio ( $p$ ), mínimo ( $m_i$ ), y máximo ( $m_a$ ) de acciones.

La función busca la combinación para maximizar la ganancia utilizando un enfoque iterativo basado en un stack. Tenemos unas variables donde `maxGanancia` almacena la ganancia máxima encontrada hasta el momento, `size` el número total de tripletas en la lista, `stack` nos representa el espacio de búsqueda en donde cada elemento es un arreglo `int` que contiene un `index` donde será el índice actual, `acciones` que es el número de acciones acumuladas y la ganancia que será la acumulada hasta este estado.

Se tiene un bucle de Búsqueda donde se itera hasta que el stack esté vacío y en cada iteración, se desapila el último estado (`current`) para procesar. Si el número de acciones (`acciones`) no supera el límite ( $A$ ), se actualiza la `maxGanancia` con el mayor valor entre la ganancia acumulada y la ganancia máxima conocida. Hacemos una validación en donde si el número de acciones (`acciones`) no supera el límite ( $A$ ), se actualiza la `maxGanancia` con el mayor valor entre la ganancia acumulada y la ganancia máxima.



## **Clase Tripletta**

Se encapsula la información de una oferta o un intervalo de acciones en el problema. Cada instancia de `Tripletta` contiene tres atributos,  $p$  representa el precio o la ganancia por cada acción seleccionada,  $m_i$  es el número mínimo de acciones que se pueden tomar y  $m_a$  es el número máximo de acciones. En el constructor se inicializan los atributos con los valores proporcionados al momento de crear la instancia de la clase. Con los métodos `getter` y `setter` permitimos acceder y modificar los atributos de forma controlada.

## **Interfaz**



Esta interfaz permite al usuario gestionar oferentes, configurar parámetros, seleccionar algoritmos y ejecutar cálculos relacionados con subastas, todo dentro de un diseño visual moderno y dinámico.

### Complejidad Computacional TERMINAL

Costo de avanzar	1
Costo de borrar	2
Costo de reemplazar	3
Costo de insertar	2
Costo de eliminar hasta el final	1

Se realizan mediciones de tiempo con al menos 5 casos de crecimiento

### **Solución de fuerza bruta problema de Terminal inteligente**

**1.**

**Cadena inicial** = rescue

**Cadena destino** = secure

**El tiempo de ejecución fue de:** 0.043534875 segundos

**2. Cadena inicial** = earth

**Cadena destino** = heart

**El tiempo de ejecución fue de:** 0.069489792 segundos

**3.**

**Cadena inicial** = francesa

**Cadena destino** = ancestro

**El tiempo de ejecución fue de:** 1.867645959 segundos

**4. Cadena inicial** = ingenioso

**Cadena destino** = ingeniero

**El tiempo de ejecución fue de:** 4.69221025 segundos

**5. Cadena inicial** = algorithm

**Cadena destino** = altruistic

**El tiempo de ejecución fue de:** 23.114518291 segundos

### **Solución de programación dinámica de Terminal inteligente**

**1.**

**Cadena inicial** = rescue

**Cadena destino** = secure

**El tiempo de ejecución fue de:** 9.375E-5 segundos

**2.**

**Cadena inicial = earth**

**Cadena destino = heart**

**El tiempo de ejecución fue de: 7.959E-6 segundos**

**3.**

**Cadena inicial = francesa**

**Cadena destino = ancestro**

**El tiempo de ejecución fue de: 1.3417E-5 segundos**

**4. Cadena inicial = ingenioso**

**Cadena destino = ingeniero**

**El tiempo de ejecución fue de: 1.5542E-5 segundos**

**5. Cadena inicial = algorithm**

**Cadena destino = altruistic**

**El tiempo de ejecución fue de: 1.6792E-5 segundos**

## **Solución voraz Terminal Inteligente**

**1.**

**Cadena inicial = rescue**

**Cadena destino = secure**

**El tiempo de ejecución fue de: 1.9150831999999995E-4 segundos**

**2.**

**Cadena inicial = earth**

**Cadena destino = heart**

**El tiempo de ejecución fue de: 1.4634159999999995E-5 segundos**

**3. Cadena inicial = francesa**

**Cadena destino = ancestro**

**El tiempo de ejecución fue de: 8.52512E-6 segundos**

**4. Cadena inicial = ingenioso**

**Cadena destino = ingeniero**

**El tiempo de ejecución fue de: 8.990660000000002E-6 segundos**

5. Cadena inicial = algorithm  
Cadena destino = altruistic  
El tiempo de ejecución fue de: 1.780498E-5 segundos

## Subasta Dinámica

**Caso 1 (500, 100, 600)**

**Resultado:** 300000

**Tiempo de ejecución:** 0.006288583 segundos

**Caso 2 (450, 400, 800)**

**Resultado:** 360000

**Tiempo de ejecución:** 0.005158333 segundos

**Caso 3 (100, 0, 1000)**

**Resultado:** 100000

**Tiempo de ejecución:** 0.002839042 segundos

**Caso 4 (200, 100, 300) (300, 200, 400)**

**Resultado:** 180000

**Tiempo de ejecución:** 0.001098 segundos

**Caso 5 (100, 50, 150) (200, 150, 300) (300, 200, 400)**

**Resultado:** 195000

**Tiempo de ejecución:** 4.69083E-4 segundos

## Subasta Bruta

**Caso 1 - Tiempo de Ejecución:**

**Tiempo:** 0.00513275 segundos

**Caso 2 - Tiempo de Ejecución:**

**Tiempo:** 0.003482208 segundos

**Caso 3 - Tiempo de Ejecución:**

**Tiempo:** 1.5375E-4 segundos

**Caso 4 - Tiempo de Ejecución:**

**Tiempo:** 3.27459E-4 segundos

**Caso 5 - Tiempo de Ejecución:**

**Tiempo:** 0.386749708 segundos

**Configuración de los 5 casos de prueba**

Tripleta t1 = new Tripleta(50, 20, 50);

Tripleta t2 = new Tripleta(39, 150, 200);

Tripleta t3 = new Tripleta(40, 120, 130);

Tripleta t4 = new Tripleta(100, 40, 70);

Tripleta t5 = new Tripleta(60, 30, 90);

List<Tripleta> caso1 = List.of(t1, t2, t3);

List<Tripleta> caso2 = List.of(t3, t4, t5);

List<Tripleta> caso3 = List.of(t1, t4);

List<Tripleta> caso4 = List.of(t2, t5);

List<Tripleta> caso5 = List.of(t1, t2, t3, t4, t5);

**Subasta Voraz**

**Tiempos de ejecución:**

**Caso 1 - Tiempo:** 0.002464083 segundos

**Caso 2 - Tiempo:** 1.3875E-5 segundos

**Caso 3 - Tiempo:** 5.667E-6 segundos

**Caso 4 - Tiempo:** 4.334E-6 segundos

**Caso 5 - Tiempo:** 8.584E-6 segundos

**Configuración de los 5 casos de prueba**

Tripleta t1 = new Tripleta(50, 20, 50);

Tripleta t2 = new Tripleta(39, 150, 200);

Tripleta t3 = new Tripleta(40, 120, 130);

Tripleta t4 = new Tripleta(100, 40, 70);

Tripleta t5 = new Tripleta(60, 30, 90);

List<Tripleta> caso1 = List.of(t1, t2, t3);

List<Tripleta> caso2 = List.of(t3, t4, t5);

List<Tripleta> caso3 = List.of(t1, t4);



```
List<Tripleta> caso4 = List.of(t2, t5);  
List<Tripleta> caso5 = List.of(t1, t2, t3, t4, t5);
```

Se hace el promedio de las 50 ejecuciones en la terminal y la subasta, dando los resultados siguientes:

### Terminal Fuerza bruta

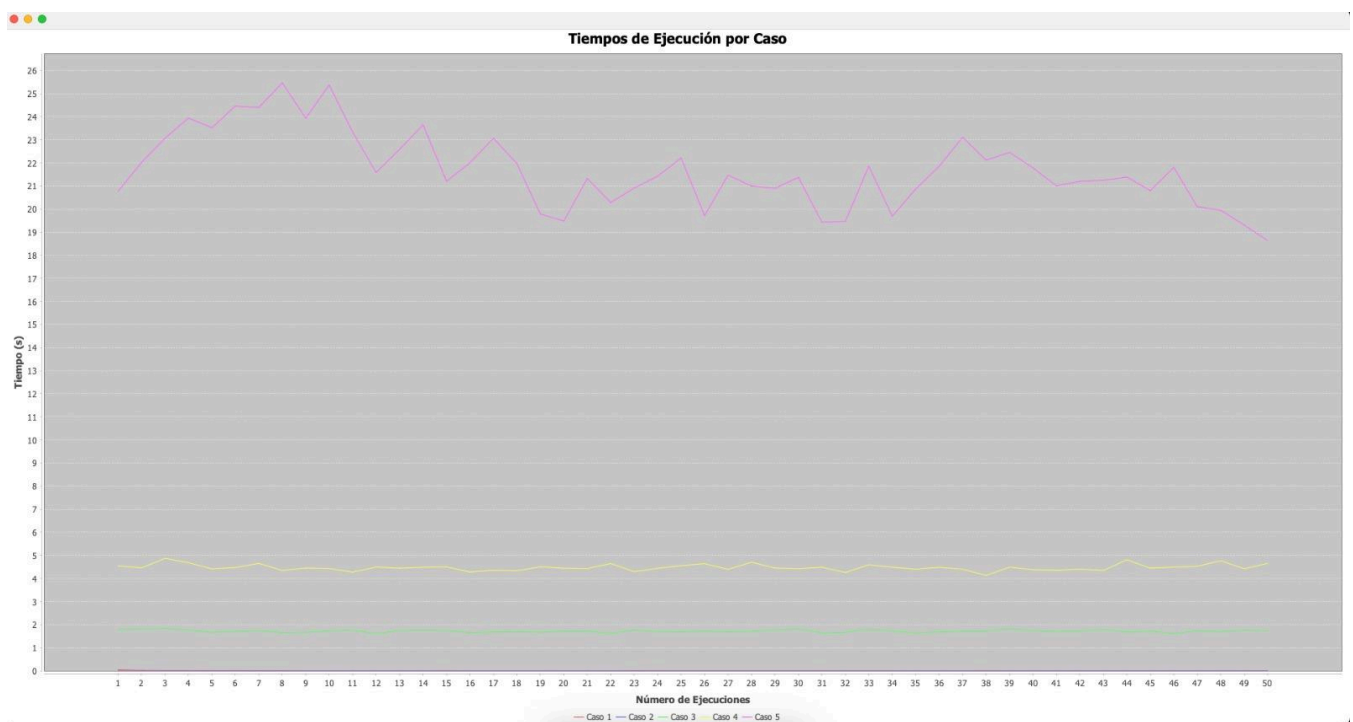
**Caso 1 - Promedio:** 0.009371502499999998 segundos

**Caso 2 - Promedio:** 7.9230922000000001E-4 segundos

**Caso 3 - Promedio:** 1.7198328124200004 segundos

**Caso 4 - Promedio:** 4.475695381699998 segundos

**Caso 5 - Promedio:** 21.685962742500006 segundos



### Terminal Dinámica

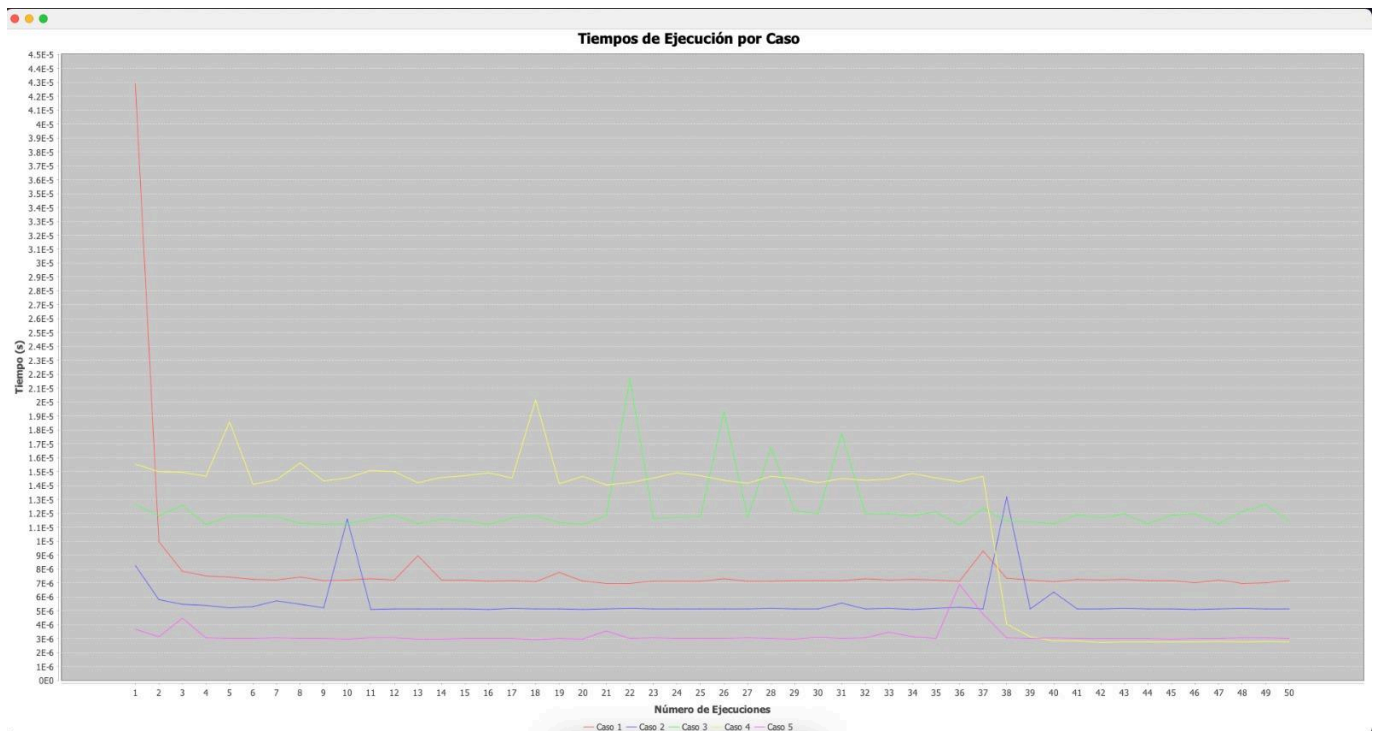
**Caso 1 - Promedio:** 8.054039999999997E-6 segundos

**Caso 2 - Promedio:** 5.564159999999999E-6 segundos

**Caso 3 - Promedio:** 1.2280899999999998E-5 segundos

**Caso 4 - Promedio:** 1.1748260000000006E-5 segundos

**Caso 5 - Promedio:** 3.18414E-6 segundos



## Terminal Voraz

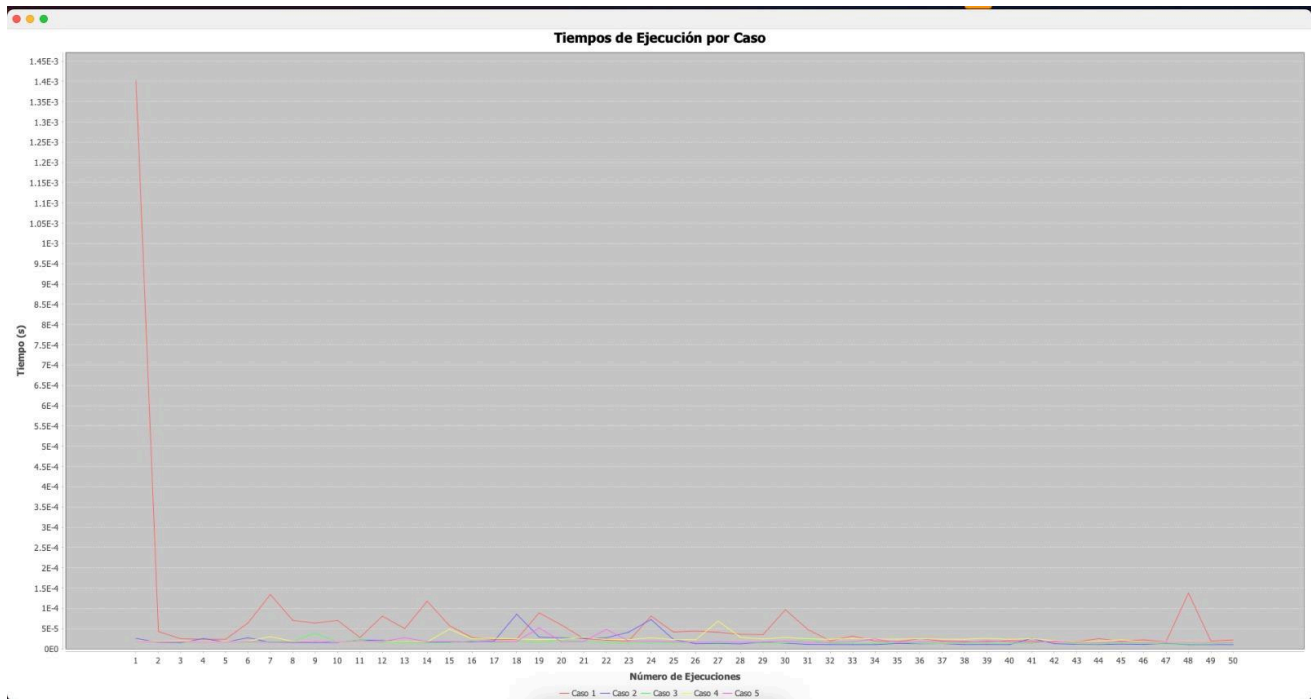
**Caso 1 - Promedio:** 6.895923999999999E-5 segundos

**Caso 2 - Promedio:** 1.8813480000000004E-5 segundos

**Caso 3 - Promedio:** 1.582248E-5 segundos

**Caso 4 - Promedio:** 2.298834E-5 segundos

**Caso 5 - Promedio:** 1.855402E-5 segundos



## Subasta Pública Bruta

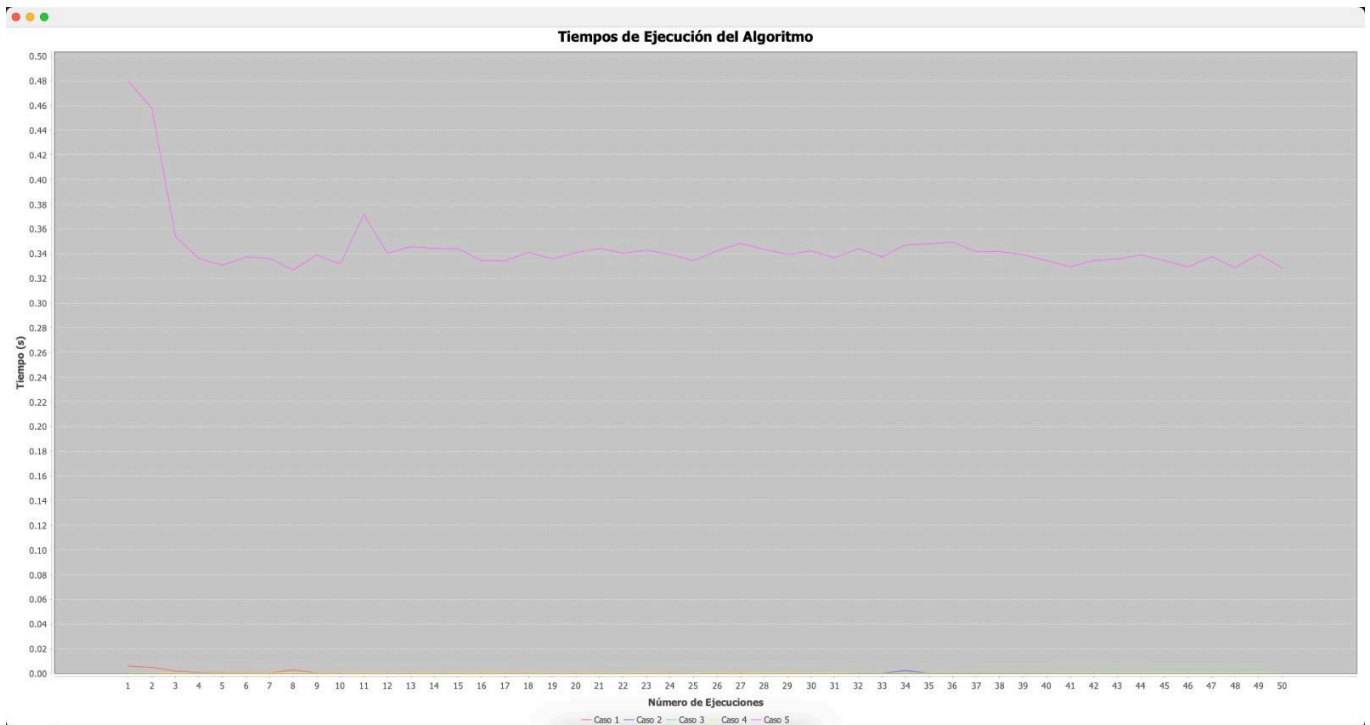
**Caso 1 - Promedio:** 6.494783599999999E-4 segundos

**Caso 2 - Promedio:** 2.7468580000000005E-4 segundos

**Caso 3 - Promedio:** 1.3192504E-4 segundos

**Caso 4 - Promedio:** 1.7755169999999994E-4 segundos

**Caso 5 - Promedio:** 0.34464582826000006 segundos



## Subasta Pública Dinámica

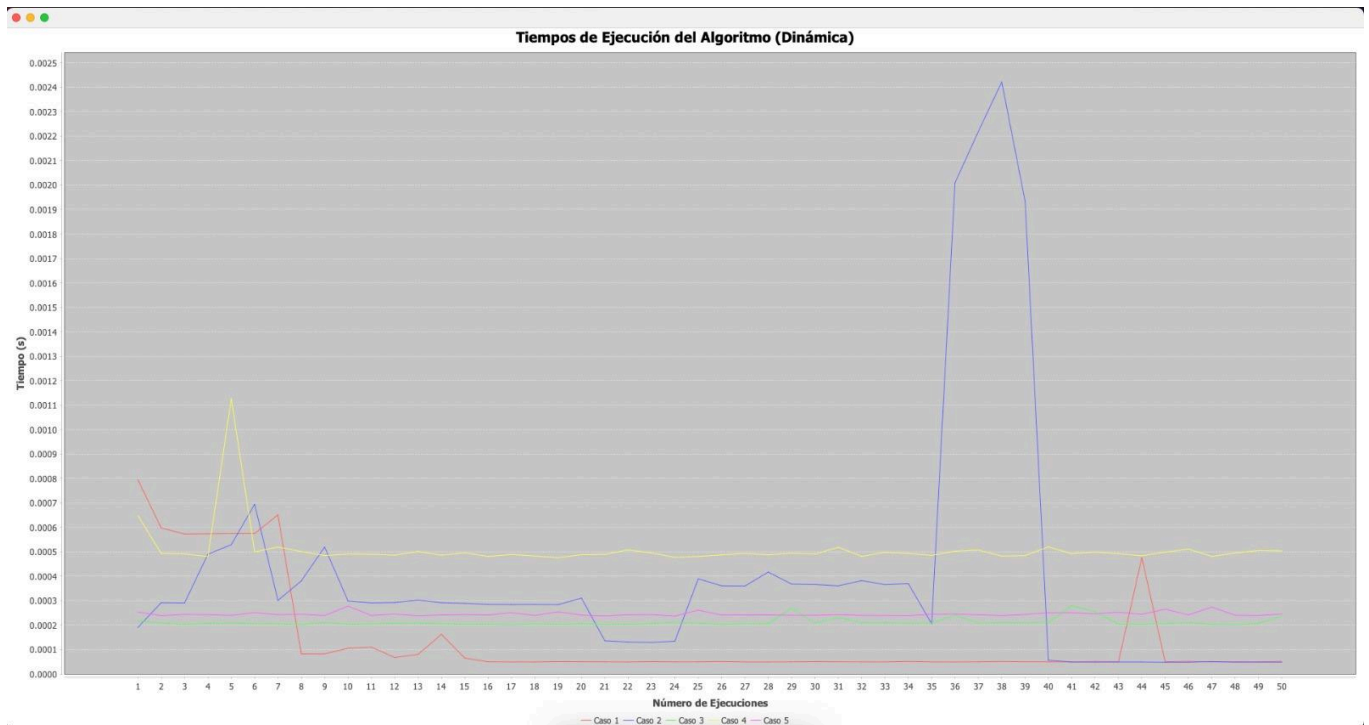
**Caso 1 - Promedio:** 1.4555502000000000E-4 segundos

**Caso 2 - Promedio:** 4.1023831999999997E-4 segundos

**Caso 3 - Promedio:** 2.1223668E-4 segundos

**Caso 4 - Promedio:** 5.0899833999999998E-4 segundos

**Caso 5 - Promedio:** 2.4536662E-4 segundos



## Subasta Pública Voraz

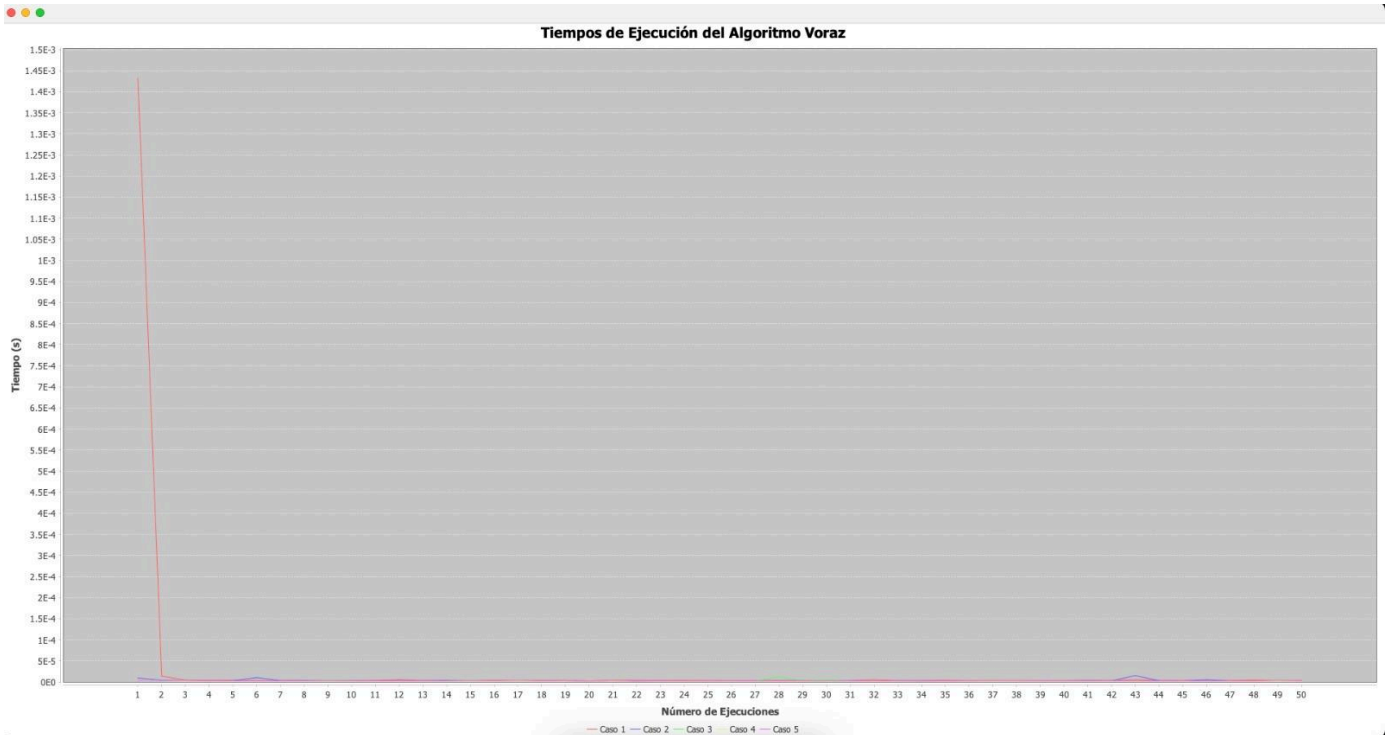
**Caso 1 - Promedio:** 3.2351640000000007E-5 segundos

**Caso 2 - Promedio:** 3.40252E-6 segundos

**Caso 3 - Promedio:** 2.8667400000000007E-6 segundos

**Caso 4 - Promedio:** 2.09162E-6 segundos

**Caso 5 - Promedio:** 2.7250000000000001E-6 segundos



## Informe de complejidades

### Terminal

**Fuerza Bruta**  $O(5^{\max(n,m)})$  n es la longitud de la cadena inicial y m es la longitud de la cadena final, es 5 porque se puede generar hasta 5 hijos por nodo ya que es la cantidad total de operaciones.

**Programación Dinámica:** Se tiene  $(m + 1) * (n + 1)$  ya que se tiene una fila y columna adicional por los casos bases esto muestra que la complejidad es  $O(m \times n)$  donde m y n son las longitudes de inicio y destino.

**Programación Voraz:** Es lineal con respecto a la longitud de las cadenas es decir  $O(n + m)$  donde m y n son las longitudes de inicio y destino, se recorre una sola vez.

### Subasta

**Fuerza Bruta:**  $O(R^n)$  (donde R es el rango promedio de acciones posibles por tripleta

Ma-Mi+1, y n el número de elementos en tripletaList)

**Programación Dinámica:**  $O(T * A * R)$  T es el número de tripletas, A el número de acciones y R el rango  $R = Ma - Mi + 1$  de la tripleta

**Programación Voraz:**  $O(n \log(n))$  debido a la ordenación de tripletas en este enfoque