Universidad del valle

Escuela de Ingeniería de Sistemas y Ciencias de la Computación

El problema del Zoológico de Cali

Profesor:
Jesus Alexander Aranda

Estudiantes:
Nicol Ortíz (2241463), Carlos Tovar(1741699),
Heidy Mina(1931720), Laura Murillas(1944153)

Diciembre 2023

0. Instrucciones de ejecución

En la raíz del proyecto:

```
python sol[1-3]/main.py test/test[1-9].txt
```

Por ejemplo, para ejecutar el test 2 en la solución 1 se debe ejecutar:

```
python sol1/main.py test/test2.txt
```

test7.txt y test8.txt son las entradas ejemplo que están en el enunciado del proyecto.

1. Soluciones planteadas

1.1. Listas (extendidas con tuplas) y diccionarios

Complejidad de construcción de las listas y diccionarios: Lineal
Complejidad de los algoritmos usados para dar respuesta a lo requerido: Cuadrático

La idea detrás de esta solución se basa en ordenar gradualmente, comenzando por los conjuntos más pequeños, que son las escenas con los animales, esto debido a que en un principio es sencillo establecer la relación de orden entre los animales gracias a las grandezas que nos da la entrada. A medida que se ordenan las escenas, se les asigna una etiqueta que guarda la grandeza total. Esta etiqueta se utiliza para guiar el ordenamiento del siguiente conjunto, que son las partes, creando así una progresión en la solución.

Para entender más a fondo la solución, veamos una descripción de cada función que se definió para llegar al resultado.

1.1.1. Construyendo las estructuras de datos (`leer_archivo`):

En primer lugar, para construir las estructuras de datos se lee la entrada desde el archivo de prueba con la función leer_archivo, que se encarga de recorrer todas las líneas del archivo hasta llenar todas las estructuras de datos, el cual tiene 1+n+m*k líneas. Como el algoritmo se encarga de leer el archivo línea por línea de forma secuencial, su complejidad es O(n + mk).

1.1.2. Calculando las participaciones (`participacion_animales`, `animal_mas_menos_participacion`)

Estas dos funciones se usan para calcular los animales con más y menos participaciones. `participacion_animales` recorre toda la apertura, que tiene (m-1)*k escenas, y por cada escena recorre los 3 animales que la conforman. En cada aparición de un animal, suma 2 participaciones en un diccionario de participaciones. Las participaciones aumentan en 2 debido a que únicamente estamos recorriendo la apertura y se sabe que cada escena de la apertura aparece una vez en las demás partes del espectáculo. Esto nos da el resultado que necesitamos sin tener que recurrir a contar las participaciones en todas las partes. La complejidad de esta función es O(3(m-1)k), donde (m-1)*k representa el número total de elementos en la apertura.

`animal_mas_menos_participacion` usa el diccionario creado anteriormente para encontrar los animales con menos y más participaciones. En un primer recorrido, encuentra la participación máxima y mínima, lo cual tiene una complejidad lineal O(n) y luego realiza otro recorrido para obtener todos los animales con esas cantidades máximas o mínimas de participación, lo cual también es lineal.

Por lo que la complejidad de calcular encontrar los animales con más y menos participaciones es $O(3(m-1)k)+O(n) = O((m-1)k)$

1.1.3. Ordenando la escena (`ordenar_escena`)

Ordena una escena dada teniendo en cuenta las grandezas de los animales en el diccionario de grandezas, la estrategia que usa para hacer el ordenamiento es BubbleSort, la cual tiene una complejidad de O(n^2). En esta función se retorna la escena ordenada y su grandeza respectiva, la cual nos servirá para tener una relación de orden que nos permita ordenar las partes. Cabe destacar que una escena siempre tiene 3 animales, por lo que, a pesar de que el algoritmo tiene una complejidad de O(n^2), n siempre será 3, por lo que la función es O(1)

1.1.4. Ordenando la parte (`ordenar_parte`)

Ordena una parte compuesta de escenas. En primer lugar debe ordenar todas sus escenas, lo cual es constante en cada escena. Se hace un recorrido de toda la parte para ordenarlas y obtener la relación de orden que requerimos para ordenarla. Este algoritmo también usa la estrategia de BubbleSort, la cual tiene una complejidad O(n^2). Cabe destacar que, aunque ordenar cada escena es constante, una parte tiene k escenas por lo que la complejidad es O(k^2)

1.1.5. Ordenando las partes (`ordenar_partes`)

Ordena todas las partes del espectáculo (sin incluir la apertura que es una parte que se ordena por separado). Para ello, se debe ordenar primero cada una de sus partes con la función `ordenar_parte` con lo cual se obtiene la relación de orden entre las partes para poder ordenarlas. La estrategia de ordenamiento en este algoritmo también es BubbleSort que es O(n^2). En este caso, la cantidad de partes está dada por m-1, y ordenar cada una de ellas es O(k^2) por lo que la complejidad de este algoritmos es O((m-1)*k^2)

1.1.6. Promedio

El promedio de las grandezas se puede calcular de forma constante O(1) dividiendo la grandeza de total de la apertura entre (m-1)*k

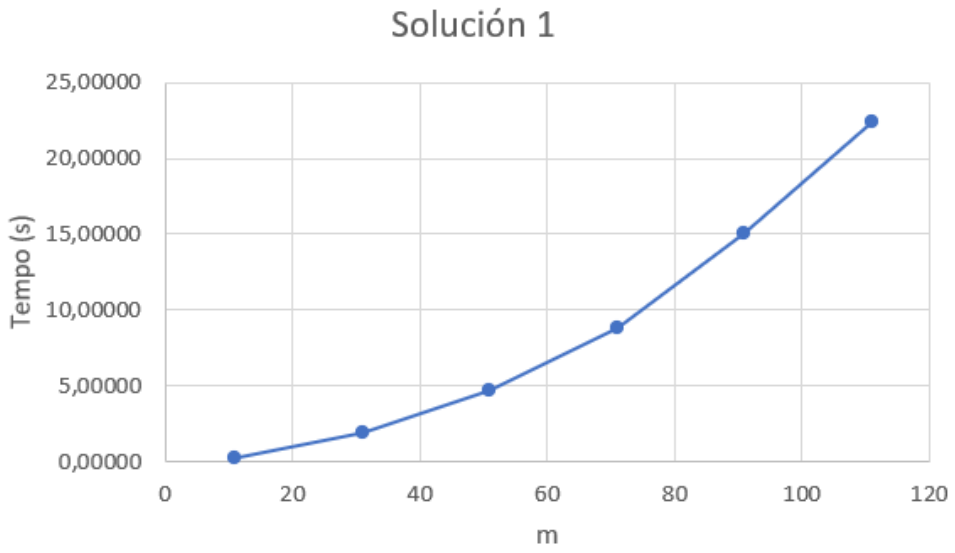
1.1.7. La solución

Para ordenar el espectáculo completo, se hace uso de la función `ordenar_parte` para ordenar la apertura, la función `ordenar_partes` para ordenar todas las partes, `min_max_participacion` para encontrar los animales con menos y más participaciones. Y para encontrar las escenas con menor y mayor grandeza, se accede a la primera y la última escena de la apertura ya ordenada. Entonces:

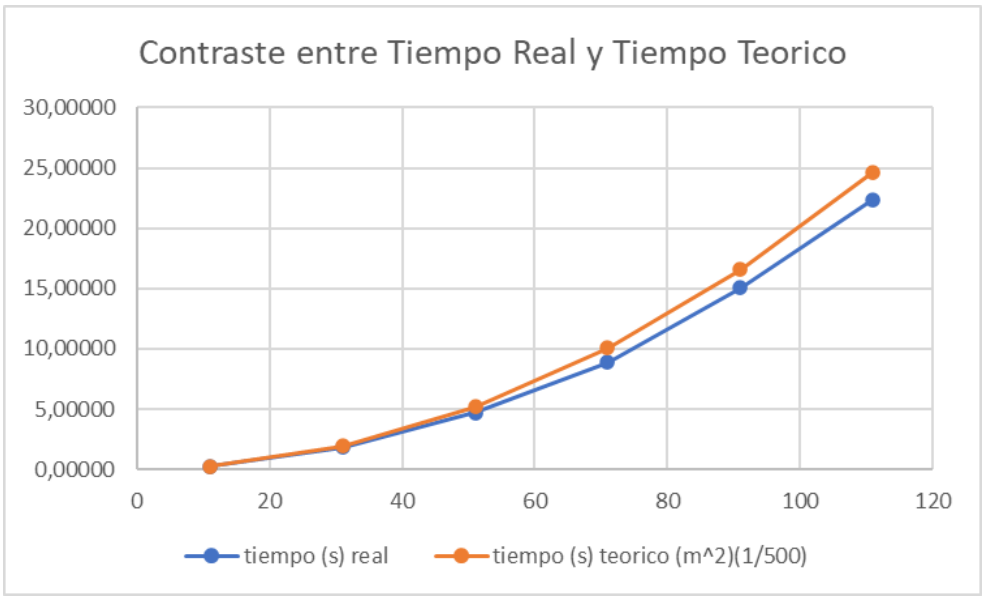
leer archivo: O(n+mk)
participaciones: O((m)k)
ordenar apertura: O(n^2)
ordenar partes: O((m)*k^2)
promedio: O(1)

por lo cual, la solución es O(n^2)
Se hicieron pruebas con distintas entradas variando m y dejando fijos n y k y se tomaron los tiempos de ejecución en cada caso para graficar el comportamiento de la solución.

Test	n	m	k	tiempo (s)
Test1	50	11	100	0,23969
Test2	50	31	100	1,85374
Test3	50	51	100	4,71986
Test4	50	71	100	8,85030
Test5	50	91	100	15,04764
Test6	50	111	100	22,36014



Para hacer el contraste entre el tiempo real y el tiempo teórico, se asumió una función dada por la complejidad ya mencionada, multiplicada por la constante (1/500). Se usa esta constante a conveniencia para hacer una comparación más ajustada, lo cual es válido teniendo en cuenta que en la complejidad dada por la notación asintótica O(cf(n)) = O(f(n)) donde c es alguna constante.



1.2. Listas enlazadas y diccionarios

Complejidad de construcción de las listas enlazadas: O(n)
Complejidad de los algoritmos usados para dar respuesta a lo requerido: O(nlog n) y O(n) y O(1).

1.2.1. Función leer_archivo(filename):

La función **leer_archivo** se encarga de leer un archivo específico y crear estructuras de datos basadas en la información del archivo. Toma un parámetro **filename**, abre el archivo y extrae datos para construir un diccionario de animales con sus respectivas grandezas, una lista enlazada de listas que representan escenas (**apertura**), y otra lista enlazada de listas que contienen partes de escenas (**partes**). Finalmente, la función devuelve las variables **n, m, k, animales, apertura, partes**.

1.2.2. Clase Nodo

La clase **Nodo** representa los nodos que componen una lista enlazada. Cada nodo tiene un dato, una referencia al siguiente nodo y una propiedad de grandeza asociada al dato.

1.2.3. Clase LinkedList

La clase **LinkedList** es una estructura de datos que implementa una lista enlazada. Contiene métodos para manejar la lista, como obtener y establecer la cabeza, así como insertar elementos.

1.2.4. Funciones de manipulación de listas enlazadas

- **recorrerLista(head, mensaje)**: Imprime todos los elementos de una lista enlazada, incluyendo listas enlazadas anidadas, utilizando un enfoque recursivo para recorrer la estructura.
- **contarRepetidos(linked_list)**: Cuenta cuántas veces se repite un animal en las escenas de una lista enlazada, incluyendo listas enlazadas anidadas y devuelve un diccionario con los animales y su participación.
- **hallarRepetidos(apertura, bool)**: Encuentra los animales que más o menos participaron en escenas, dependiendo del booleano proporcionado, devolviendo una lista con los animales encontrados y su número de repeticiones.
- **get_max_value(input)**: Obtiene el valor máximo de una lista enlazada.

1.2.5. Funciones del algoritmo Merge Sort

- **split_list(head)**: Divide la lista enlazada en dos mitades.
- **merge(left, right)**: Combina dos listas enlazadas ordenadas en una sola lista enlazada ordenada.
- **merge_sort(head)**: Algoritmo principal de Merge Sort para ordenar una lista enlazada.

1.2.6. Otras funciones

- **merge_sort_parte(apertura)**, **merge_sort_escena_parte(partes)**, **merge_sort_animales_parte(partes)**: Aplican el algoritmo Merge Sort a diferentes partes de las estructuras de datos.
- **menorEscena(apertura)**, **mayorEscena(apertura)**: Encuentran la escena con la menor y mayor grandeza, respectivamente.
- **promedioGrandeza(linked_list)**: Calcula el promedio de las grandezas en una lista enlazada.
- **main(filename)**: La función principal que realiza la lógica principal del programa utilizando todas las funciones anteriores. Lee el archivo, realiza operaciones en las estructuras de datos, imprime resultados y mide el tiempo de ejecución.

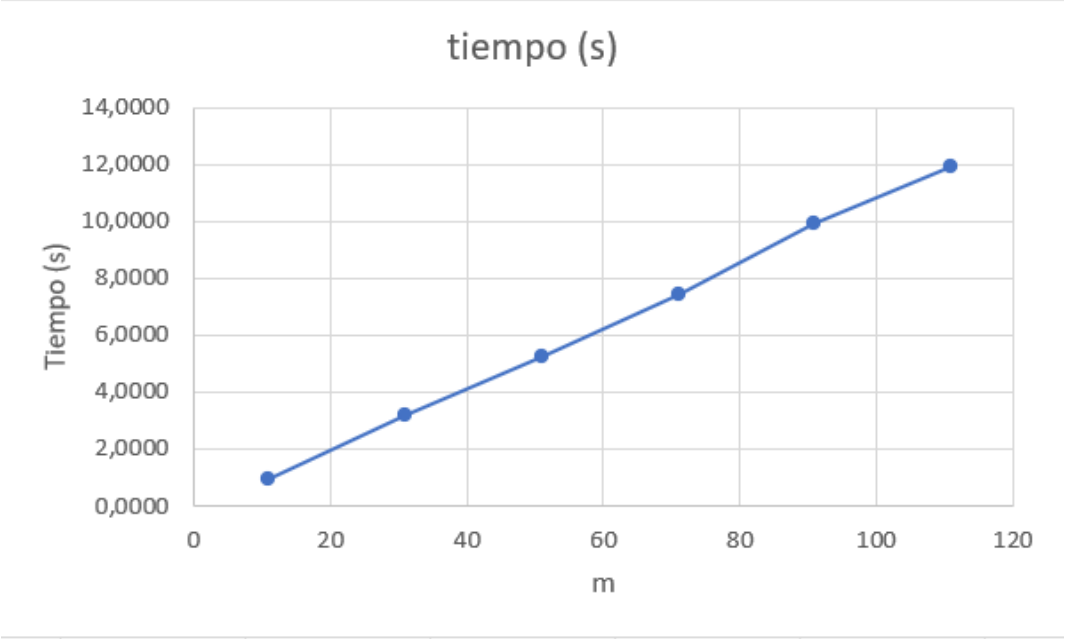
1.2.7. Complejidades.

- leer_archivo(filename), O(n)
- traverse_nested_linked_lists(head, mensaje), O(n)
- contarRepetidos(linked_list), O(n)
- hallarRepetidos(apertura, bool), O(n+m)
- get_max_value(input), O(n)
- split_list(head), O(n)
- merge(left, right), O(n+m)
- merge_sort(head), O(nlogn)
- merge_sort_parte(apertura), O(nlogn)
- merge_sort_escena_parte(partes), O(nlogn)
- merge_sort_animales_parte(partes), O(nlogn)
- menorEscena(apertura), O(1)
- mayorEscena(apertura), O(n)
- promedioGrandeza(linked_list), O(n)
- main(filename)

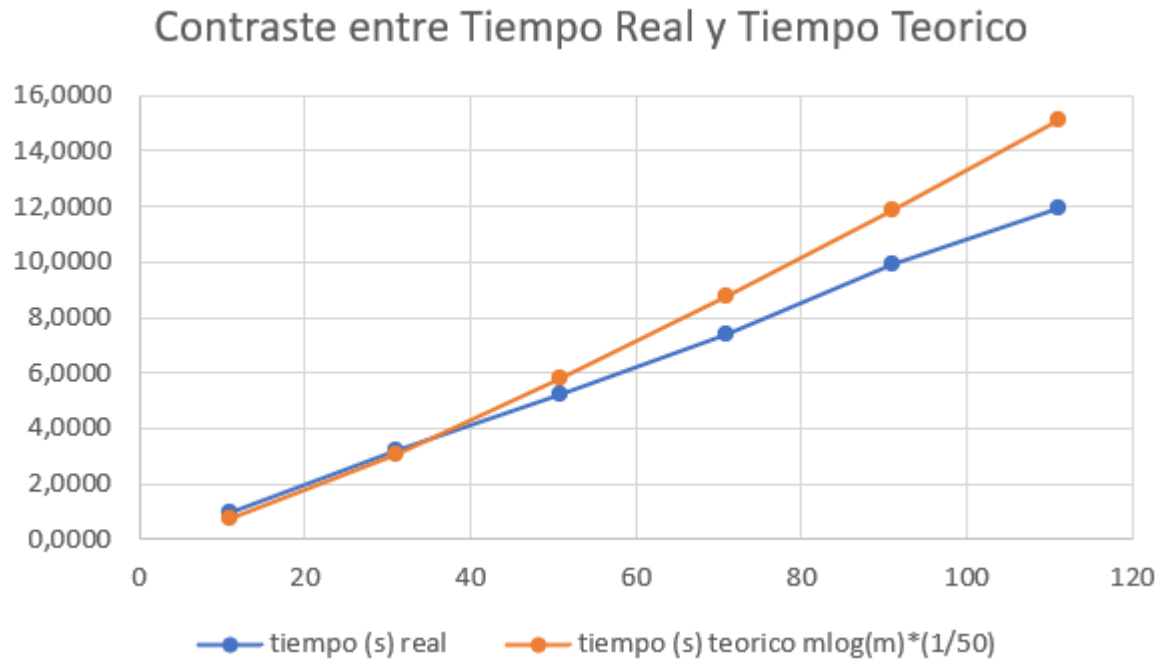
Complejidad de la solución O(nlogn)

Se hicieron pruebas con distintas entradas variando m y dejando fijos n y k y se tomaron los tiempos de ejecución en cada caso para graficar el comportamiento de la solución.

Test	n	m	k	tiempo (s)
Test1	50	11	100	0,9519
Test2	50	31	100	3,2103
Test3	50	51	100	5,2502
Test4	50	71	100	7,4182
Test5	50	91	100	9,9238
Test6	50	111	100	11,9275



Para hacer el contraste entre el tiempo real y el tiempo teórico, se asumió una función dada por la complejidad ya mencionada, multiplicada por la constante (1/50). Se usa esta constante a conveniencia para hacer una comparación más ajustada, lo cual es válido teniendo en cuenta que en la complejidad dada por la notación asintótica O(cf(n)) = O(f(n)) donde c es alguna constante.



1.3. Árboles “rojo y negro” y diccionarios.

Complejidad de construcción del árbol rojo y negro: O(nlogn)
Complejidad de los algoritmos usados para dar respuesta a lo requerido: O(1), O(logn) y O(n)

La idea central de la solución consiste en descomponer el archivo obtenido para extraer los valores de n, m, k, las características de los animales participantes, la apertura y las partes. Para implementar esta solución, se han creado clases como Animal, Escena, Apertura, Partes y Espectáculo. Estas clases se utilizaron para generar objetos a partir de los datos extraídos del archivo.

Todas las clases, a excepción de la clase Animal que solo guarda el nombre del animal y su grandeza, cuentan con atributos para almacenar varios datos. Por ejemplo, en el caso de las Escenas, este atributo almacena animales; en el caso de las Partes, guarda las (m-1)*k escenas para la apertura o k escenas para las m-1 partes. Respecto al Espectáculo, este atributo almacena las m-1 partes de todo el show.

Resulta evidente que este atributo debería hacer uso de una estructura de datos. Considerando las preguntas que se deben responder, como identificar la escena con la mayor grandeza o las partes con la mayor grandeza, así como ordenar cada escena y parte en función de su grandeza de forma ascendente, se decidió almacenar estos objetos en árboles rojo y negro. Esta elección se basa en las características de dichos árboles, que parecían ser una opción óptima, ya que para algunas de las preguntas planteadas no era necesario recorrer todos los datos en su totalidad.

1.3.1. Clase Animal, Escena, Parte y Espectáculo.

La clase Animal es la más sencilla. Está compuesta por atributos como “nombre_animal” de tipo string, “grandeza” de tipo entero, y “tipoObjeto”, el cual almacena el nombre de la clase correspondiente al objeto. Cuenta con los métodos “getNombreAnimal”, “getNombreGrandeza”, “getTipoGrandeza” y getMaximo() que permiten acceder al valor de cada atributo, cada uno de estos métodos tiene complejidad O(1).

Las clases restantes, como Escena, Parte y Espectáculo, son más similares entre sí, ya que cada una debe guardar una estructura de datos compuesta.

Atributos de estas clases.

- El atributo para almacenar animales, escenas o partes, independientemente del nombre que posea dependiendo de la clase, siempre guarda un objeto ArbolRN del cual se hablará posteriormente, para almacenar los datos en un árbol rojo y negro.
- El atributo "grandeza", un entero que guarda la suma de las grandezas de los objetos, ya sea animal, escena, parte o espectáculo.
- El atributo "tipoObjeto", almacena un string correspondiente al nombre de la clase. Esto se hace con el objetivo de identificar a qué clase pertenece un objeto. A
- Atributo "máximo", que almacena cuál es el valor con mayor grandeza en el árbol.
- Por otro lado, las clases Parte y Espectáculo tienen un atributo llamado "totalEscenas" o "totalPartes", respectivamente, para almacenar el total de nodos de sus árboles rojo y negro.

En relación a los métodos asociados, se encuentra el procedimiento de inclusión de elementos en el árbol rojo y negro que alberga el atributo mencionado previamente el cual tiene una complejidad igual al método TREE_INSERT. Las clases con el método "agregar*", se ocupa de incorporar nodos al árbol rojo y negro. Además de esta tarea, aumenta la magnitud del objeto por cada nodo añadido al árbol y en el caso específico de las clases Parte y Espectáculo, también incrementa en 1 el atributo que registra el total de nodos en el árbol.

El método "aumentar_grandeza" con complejidad O(1) incrementa el atributo "grandeza" con el entero suministrado. “getMaximo” tiene una complejidad igual a la complejidad del método TREE_MAXIMUM del árbol. Por otra parte, "getPromedioGrandeza" de la clase Partes realiza una división entre el atributo "grandeza" y el total de escenas, con el fin de calcular el promedio de una parte, su complejidad es O(1).

Los métodos que comienzan con "print*" utilizan una de las funcionalidades del árbol para imprimir los elementos de manera específica y ordenada, su complejidad es igual a la complejidad del método print_tree. En cuanto a los restantes métodos, como "getAnimales", "getEscenas", "getPartes", "getGrandeza" y "getTipoObjeto", su propósito es acceder al valor de los atributos correspondientes, por los tanto tienen complejidad O(1).

1.3.2. Clase Nodo.

La clase Nodo contiene los siguientes atributos:

valor, para almacenar el objeto ya sea de tipo Animal, Escena, Parte o Espectáculo.

hijo_izq, para almacenar al hijo izquierdo del nodo.

hijo_der, para almacenar al hijo derecho del nodo.

padre, para almacenar al padre del nodo.

color, para almacenar el color del nodo ya sea rojo o negro.

Los métodos getValor, getHijoIzq, getHijoDer, getPadre y getColor son usados para acceder al dato que alberga el atributo. Por último los métodos setValor, setHijoIzq, setHijoDer, setPadre y setColor están para modificar el valor de los respectivos atributos con el valor ingresado.
Cada unos de estos métodos tiene complejidad O(1)

1.3.3. Clase Arbol.

La clase Árbol consta de solo un atributo llamado raíz, el cual se encarga de almacenar un nodo y será la raíz del árbol.

La mayoría de los métodos de esta clase ya han sido establecidos, puesto que los árboles “rojo y negro” son estructuras previamente estudiadas.

- **Método getRaiz con complejidad O(1):** el cual solo permite acceder al valor de la raíz.
- **Método TREE_INSERT con complejidad O(nlogn):**

  El método TREE_INSERT se encarga de añadir nodos al árbol rojo y negro inicialmente vacío. Este método recibe un nodo y lo inserta en la posición correspondiente, teniendo en cuenta las características de un árbol rojo y negro. Para ello, utiliza la grandeza de cada objeto almacenado en el atributo 'valor' de la clase Nodo. Este método está especialmente diseñado para atender las necesidades específicas del problema.

  Cuando se ingresa un nodo, lo primero que hace el método es crear dos variables, x e y. Posteriormente, se ejecuta un condicional que solamente procede si el nodo a ingresar tiene como valor a un objeto Animal. Esta parte del código construye un diccionario donde las claves son los nombres de los animales y los valores representan la cantidad de veces que cada animal aparece en el árbol de nodos. Este enfoque facilita la respuesta a las preguntas relacionadas con los animales que menos y más se repiten, ya que se realiza un conteo a medida que se ingresan al árbol.

  Luego, se busca la posición correspondiente del nodo en el árbol. Los nodos ubicados a la izquierda serán aquellos con una grandeza menor. En caso de que dos nodos tengan la misma grandeza, el criterio de ubicación dependerá del valor que arroje el método 'getMaximo', el cual está presente en todas las clases.

  Una vez que el nodo se ha posicionado en el árbol, se pasa al método 'insert_fixup' para asegurar que se conserven las propiedades de un árbol rojo y negro.

  La complejidad del método TREE_INSERT es O(nlogn). En el peor caso, insertar un nodo tiene una complejidad de O(log n). Por lo tanto, si se deben ingresar “n” nodos, la complejidad total sería O(n log n).

- **Métodos LEFT_ROTATE y RIGHT_ROTATE con complejidad O(1):**

  Ambos métodos tienen el propósito de rotar nodos en una dirección específica, reorganizando las relaciones entre los nodos del árbol. Aunque implementan acciones diferentes según la rotación (derecha o izquierda), comparten una estructura similar y desempeñan un papel clave en la reestructuración del árbol. Ambos métodos tienen una complejidad constante de O(1) ya que realizan un conjunto fijo de operaciones independientemente del tamaño del árbol.

- **Método insert_fixup con complejidad O(logn):**

  Es usado después de la inserción de un nodo. Se encarga de restaurar el balance del árbol “rojo y negro” y garantizar que se cumplan las reglas definidas para este tipo de estructura.

  En líneas generales, el algoritmo realiza una serie de rotaciones y ajustes de colores en el árbol con el fin de mantener o recuperar las características necesarias para preservar las propiedades del árbol rojo y negro. El método verifica y ajusta los colores de los nodos y realiza rotaciones a lo largo del camino hacia la raíz del árbol para restablecer el equilibrio y mantener las reglas estructurales.

  La complejidad del método insert_fixup se estima en O(log n), donde "n" es la cantidad de nodos en el árbol. Esta complejidad surge debido a que el algoritmo realiza un número limitado de operaciones mientras recorre el camino desde el nodo insertado hasta la raíz, y esta distancia está limitada por la altura del árbol, que es logarítmica en función de la cantidad de nodos.

- **Métodos TREE_MAXIMUM y TREE_MINIMUN con complejidad O(logn):**

  Los métodos TREE_MAXIMUM y TREE_MINIMUM tienen como objetivo encontrar el nodo con el valor más grande y más pequeño, respectivamente, en un árbol rojo y negro. Ambos métodos operan recorriendo los nodos del árbol de manera iterativa para llegar al extremo derecho (para TREE_MAXIMUM) o al extremo izquierdo (para TREE_MINIMUM) del árbol, donde se encuentran los valores máximos y mínimos.

  El método TREE_MAXIMUM comienza desde la raíz del árbol y avanza por los hijos derechos hasta llegar al nodo más a la derecha, que contendría el valor máximo en el árbol. Por otro lado, el método TREE_MINIMUM comienza desde la raíz y avanza por los hijos izquierdos hasta llegar al nodo más a la izquierda, que contendría el valor mínimo.

  La complejidad de ambos métodos es proporcional a la altura del árbol, y en un árbol rojo y negro bien balanceado, esta altura es logarítmica en función del número de nodos del árbol. Por lo tanto, la complejidad de estos métodos se aproxima a O(log n), donde "n" representa la cantidad de nodos presentes en el árbol.

- **Método print_tree y print_tree_helper con complejidad O(n):**

  Estos métodos, print_tree y _print_tree_helper, se encargan de imprimir de manera ordenada y estructurada los nodos de un árbol rojo y negro. El método principal, print_tree, llama a un método auxiliar print_tree_helper pasando la raíz del árbol y un prefijo inicialmente vacío.

  El método print_tree_helper recibe un nodo y un prefijo que se utiliza para estructurar visualmente la salida de los nodos. Se imprime la información del nodo, que incluye su valor (en este caso, la grandeza del nodo) y su color, si está definido. Luego, de manera recursiva, se llama a este método para los hijos izquierdo y derecho del nodo actual, con un prefijo modificado para mantener la estructura visual del árbol.

La complejidad de ambos métodos, print_tree y print_tree_helper, es lineal en función del número de nodos del árbol, representado por "n". Esto se debe a que cada nodo se imprime una sola vez y, en el peor caso, todos los nodos deben ser visitados para mostrar la estructura completa del árbol. Por lo tanto, la complejidad de ambos métodos es O(n), donde "n" es la cantidad total de nodos presentes en el árbol.

Para hacer el contraste entre el tiempo real y el tiempo teórico, se asumió que la función está acotada por la complejidad ya mencionada, multiplicada por la constante (1/500)

1.3.4. Funciones implementadas

- **Función INORDER con complejidad O(n):**

  Realiza un recorrido en orden en un árbol rojo y negro. Este recorrido en orden visita todos los nodos del árbol siguiendo el esquema: primero el subárbol izquierdo, luego el nodo actual y finalmente el subárbol derecho. La función toma dos parámetros, un nodo y una función, y ejecuta la función proporcionada en cada nodo del árbol respetando el orden mencionado.

  La complejidad de este algoritmo está directamente relacionada con el número total de nodos del árbol. En el caso peor, donde se deben recorrer todos los nodos del árbol, la complejidad es O(n), siendo "n" la cantidad de nodos en el árbol.

- **Función imprimir_nombre_animal con complejidad O(1):**

  Dada una escena, accede al nombre del animal y lo imprime.

- **Funciones imprimir_escena, imprimir_parte y imprimir_espectaculo con complejidad O(n):**

  Estos métodos se encargan de imprimir la estructura y los elementos de un espectáculo, que está organizado jerárquicamente en partes, escenas y animales.

  - imprimir_nombre_animal: Imprime el nombre de un animal.
  - imprimir_escena: Imprime la información de una escena, mostrando los animales que forman parte de ella y su grandeza.
  - imprimir_parte: Imprime la información de una parte del espectáculo, mostrando la grandeza total de la parte y luego detalla las escenas contenidas en esa parte, utilizando la función imprimir_escena.
  - imprimir_espectaculo: Imprime la estructura completa del espectáculo, comenzando por sus partes y luego mostrando las escenas y animales de cada parte, utilizando las funciones imprimir_parte y imprimir_escena respectivamente.

  La complejidad de estos métodos depende del tamaño de la estructura del árbol. En el peor caso, donde se deben recorrer todos los elementos del árbol, la complejidad será O(n), siendo "n" el número total de elementos (partes, escenas y animales) presentes en el espectáculo. Esto se debe a que cada método realiza un recorrido o impresión de cada uno de los elementos una sola vez.

- **Función hallarRepeticion con complejidad O(n):**

  Tiene como objetivo encontrar los elementos que más se repiten o los que menos se repiten, dependiendo del parámetro booleano **get_max**. Se recorre el diccionario de entrada para comparar los valores de cada elemento. Inicialmente, se establecen variables para el valor extremo y los elementos correspondientes.

  Dentro del bucle de iteración, se compara cada valor con el valor extremo actual. Si el valor extremo no ha sido definido o el valor actual es mayor al valor extremo (si **get_max** es verdadero) o es menor al valor extremo (si **get_max** es falso), se actualiza el valor extremo y se restablece la lista de elementos extremos con el nuevo animal.

  En el caso de que el valor actual sea igual al valor extremo, el animal se añade a la lista de elementos extremos.

  Finalmente, la función retorna la lista de elementos extremos junto con su valor extremo correspondiente.

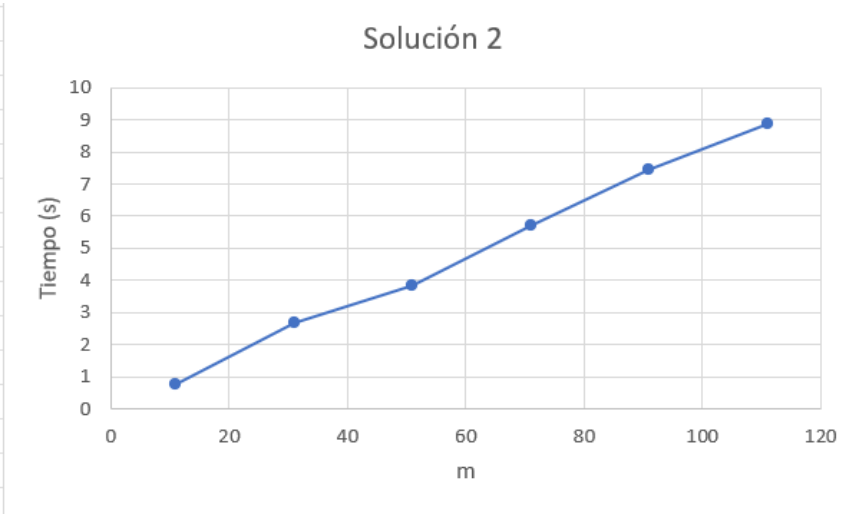
  La complejidad de este algoritmo es lineal, $O(n)$, donde n es el número de elementos en el diccionario.

1.3.5. Con lo anterior se puede decir que:

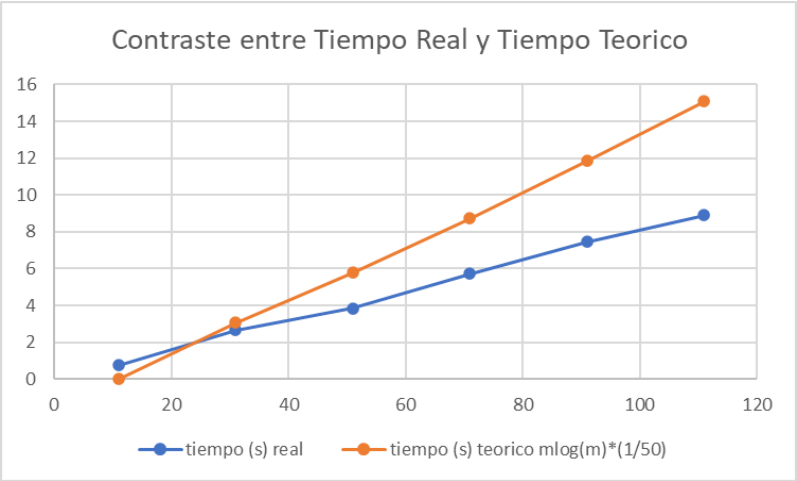
- Ordenar cada una de las escenas y partes tienen complejidad O(n) ya que el árbol "rojo y negro" se construye con un orden ya establecido, por lo tanto solo se debe hacer un recorrido inorder.
- Para hallar el elemento repetido se debe recorrer el diccionario que ya contiene las repeticiones de cada animal, así que sería O(n), teniendo en cuenta que los elementos del diccionario son mucho menores a los elementos del árbol.
- Hallar las escenas con menor y mayor grandeza tiene una complejidad O(logn) ya que esta es la complejidad de los métodos TREE_MINIMUN() y TREE_MAXIMUN().
- Hltados subrayan la importancia de considerar las características específicas del proyecto al seleccionar el algoritmo. Para conjuntos de datos pequeños, la simplicidad y rapidez de Bubble Sort pueden ser ventajosas, pero a medida que el tamaño de los datos aumenta, algoritmos como Merge Sort y la implementación con árboles muestran ser opciones más eficientes en términos de tiempo de ejecución y manejo de grandes volúmenes de información.
- Hallar el promedio de grandeza de todo el espectáculo sería constante, ya que el método getPromedioGrandeza tiene complejidad O(1).
- La solución tiene una complejidad de O(nlogn), ya que lo que más cuesta es construir el árbol.

Se hicieron pruebas con distintas entradas variando "m" y dejando fijos "n" y "k" y se tomaron los tiempos de ejecución en cada caso para graficar el comportamiento de la solución.

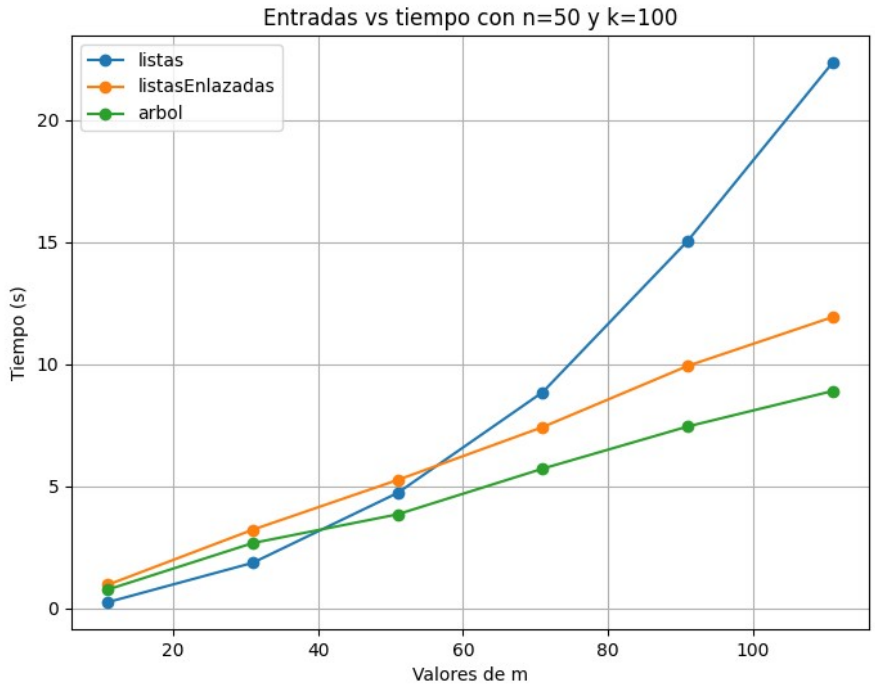
Test	n	m	k	tiempo (s)
Test1	50	11	100	0,75648
Test2	50	31	100	2,66268
Test3	50	51	100	3,84201
Test4	50	71	100	5,7132
Test5	50	91	100	7,44168
Test6	50	111	100	8,89615



Para hacer el contraste entre el tiempo real y el tiempo teórico, se asumió una función dada por la complejidad ya mencionada, multiplicada por la constante (1/50). Se usa esta constante a conveniencia para hacer una comparación más ajustada, lo cual es válido teniendo en cuenta que en la complejidad dada por la notación asintótica O(cf(n)) = O(f(n)) donde c es alguna constante.



2. Comparación de las entradas vs tiempos de las tres soluciones.



La comparación del rendimiento entre algoritmos evidencia que, para entradas pequeñas, la implementación con Bubble Sort muestra tiempos más cortos. Sin embargo, a medida que aumenta el tamaño de la entrada, Bubble Sort exhibe tiempos significativamente mayores. La implementación con listas enlazadas y Merge Sort sigue esta tendencia, mientras que la eficiencia más destacada se observa en la implementación con árboles, especialmente con entradas de mayor tamaño.

3. Prueba con un caso considerablemente grande

Prueba con n=60 m=151 k=200			
Tests	Listas (segundos)	Listas enlazadas (segundos)	Árboles (segundos)
test1	371.27546	58.48905	51.22648

Estos resultados muestran una clara diferencia en los tiempos de ejecución entre las implementaciones. La implementación con árboles ha demostrado ser la más eficiente en términos de tiempo, seguida por la implementación con listas enlazadas y finalmente por la implementación con listas.

Basándonos en estos resultados, la conclusión sería que, para este conjunto particular de parámetros (n=60, m=151, k=200), la implementación con árboles es la más eficiente en cuanto a tiempo de ejecución. Las listas enlazadas (merge sort) ocupan el segundo lugar en términos de eficiencia, siendo significativamente más rápidas que las listas convencionales para este conjunto de datos. Sin embargo, las listas convencionales (implementadas con

Bubble Sort) requirieron considerablemente más tiempo en comparación con las otras estructuras, lo que sugiere una menor eficiencia para este escenario específico de datos.

4. Conclusiones

- La implementación con árboles "rojo y negro" destaca por su eficiencia, especialmente con conjuntos de datos más grandes, gracias a la complejidad logarítmica de las operaciones en este tipo de estructuras. Sin embargo, se debe tener en cuenta que la construcción del árbol tiene una complejidad de O(n log n), que puede ser significativa para conjuntos de datos muy grandes.
- La implementación con listas enlazadas y Merge Sort muestra un rendimiento sólido, con una complejidad O(n log n) para el algoritmo de ordenamiento. Aunque puede ser menos eficiente que los árboles para ciertos tamaños de datos, su rendimiento es notable.
- La implementación con listas extendidas y Bubble Sort es la menos eficiente, con una complejidad cuadrática de O(n^2). Aunque puede ser adecuada para conjuntos de datos pequeños, su rendimiento se degrada rápidamente con entradas más grandes.
- La elección del algoritmo adecuado depende del tamaño de los datos y de la eficiencia requerida. Para conjuntos de datos pequeños, la implementación con Bubble Sort puede ser aceptable debido a su simplicidad.
- Se realizaron pruebas empíricas con diferentes conjuntos de datos para contrastar los tiempos reales con los tiempos teóricos. En general, las implementaciones con listas y listas enlazadas mostraron un buen ajuste con las complejidades teóricas, mientras que la implementación con árboles también se alineó adecuadamente con las expectativas, aunque no se ajusta tan bien como las dos anteriores.
- Las pruebas empíricas son fundamentales para comprender el rendimiento real de las soluciones y validar las complejidades teóricas.

5. Consideraciones finales

Para conjuntos de datos pequeños, se debería considerar la simplicidad de implementación y la rapidez de ejecución de la implementación con listas y BubbleSort. Para conjuntos de datos medianos a grandes, la implementación con listas y MergeSort ofrecen un buen equilibrio entre eficiencia y claridad en el código. La implementación de árboles "rojo y negro" se debería considerar cuando se necesite un rendimiento óptimo para conjuntos de datos muy grandes.