PRÁCTICA 4

Estructuras de datos

Grupo 201 GII

María Guzmán Valdezate
Guillermo López de Arechavaleta Zapatero

Contenido

Descripción y Análisis de Métodos

seleccionMultiple(List<E> lista, int [] seleccionados)

Descripción: Devuelve una nueva lista con los elementos de la lista original cuyos índices coinciden con los valores proporcionados en el array seleccionados. Si algún índice está fuera del rango de la lista, se lanza una excepción IndexOutOfBoundsException.

seleccionInversaMultiple_1(List<E> lista, int [] eliminados)

Descripción: Devuelve una nueva lista que excluye los elementos de la lista original cuyos índices coinciden con los valores proporcionados en el array eliminados. Utiliza un array auxiliar para marcar los índices eliminados y construye la lista final excluyendo los elementos marcados. Si algún índice está fuera del rango de la lista, se lanza una excepción IndexOutOfBoundsException.

seleccionInversaMultiple_2(List<E> lista, int [] eliminados)

Descripción: Devuelve una nueva lista que excluye los elementos de la lista original cuyos índices coinciden con los valores proporcionados en el array eliminados. Este método elimina los elementos en orden inverso para optimizar el proceso de eliminación. Si algún índice está fuera del rango de la lista, se lanza una excepción IndexOutOfBoundsException.

particion(List<E> lista, int [] destino)

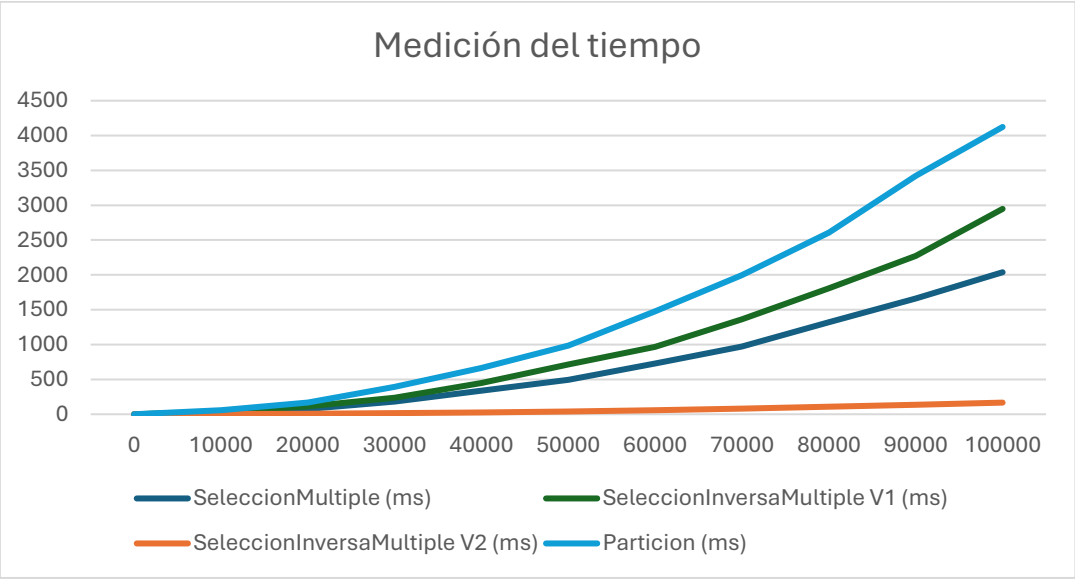
Descripción: Divide la lista original en varias sublistas según los valores proporcionados en el array destino. Cada elemento de la lista original se coloca en una sublista determinada por el valor correspondiente en el array destino. Si el valor en destino es -1, el elemento no se añade a ninguna sublista. Si la lista o el array destino son null, o si sus tamaños no coinciden, se lanza una excepción NoSuchElementException.

Pruebas y gráficas

Para validar las complejidades teóricas, se ha diseñado una clase de prueba que mide los tiempos de ejecución de los tres métodos utilizados en el programa utilizando el test facilitado y debidamente implementado TestRendimientoSeleccionListas.

Se realizan múltiples pruebas con tamaños de lista crecientes y se registran los tiempos de ejecución en milisegundos de las cuatro implementaciones de métodos pedidos. A continuación, se presentan los resultados en una tabla y sus respectivas representaciones gráficas.

Tamaño de la lista	SeleccionMultiple (ms)	SeleccionInversaMultiple V1 (ms)	SeleccionInversaMultiple V2 (ms)	Particion (ms)
0	0	0	0	0
10000	26	33	2	56
20000	77	112	6	168
30000	180	238	16	391
40000	337	449	28	666
50000	496	716	41	984
60000	728	966	59	1477
70000	971	1362	83	1996
80000	1321	1806	109	2609
90000	1659	2272	134	3422
100000	2038	2947	167	4123



Las diferencias que encontramos en los tiempos tienen que deberse a la implementación, ya que la implementación del test es igual en los cuatro métodos. Con respecto a las implementaciones, empezamos revisando el método seleccionMultiple simplemente recorre todos los elementos de un array de enteros (seleccionados) y añade a una nueva lista el elemento en la posición correspondiente de la lista original, lo que tiene una O(n), donde n es la longitud del array seleccionados.

Comparándolo con el segundo método seleccionInversaMultiple_1, vemos que este método crea un array auxiliar (lista_nums) del mismo tamaño que la lista original, inicializado con los índices de la lista. Luego, recorre el array eliminados y marca como -1 los índices que deben ser eliminados. Finalmente, recorre lista_nums y añade a una nueva lista los elementos de la lista original cuyos índices no están marcados como -1. Esto tiene una complejidad de O (n + m), donde n es la longitud de la lista y m es la longitud del array eliminados lo que se traduce en O(3*n/2) debido a que la lista eliminados tiene una longitud de lista/2 = n/2. Como se simplifica es O(n).

Por otro lado, el método seleccionInversaMultiple_2 crea una copia de la lista original y recorre el array de eliminados en orden inverso, eliminando los elementos correspondientes de la copia. Esto tiene una complejidad de O(m), donde m es la longitud del array eliminados, lo que causa una O(n/2) ya que, como hemos comentado, la longitud de eliminados es n/2. Como se simplifica es O(n).

Por último, el método particion recorre el array destino y, para cada elemento, asigna el elemento correspondiente de la lista original a una sublista específica. Si el valor en destino es -1, el elemento no se añade a ninguna sublista. Esto tiene una complejidad de O(n) en el mejor caso, pero puede acercarse a $O(n^2)$ en el peor caso debido al bucle while interno.

Como hemos visto, los valores de O () cuadran con los obtenidos en el experimento y representados en la gráfica. El único método que nos sorprendió al obtener los valores ha sido seleccionInversaMultiple_2 debido a su altísima eficiencia.

Conclusiones

seleccionMultiple es muy eficiente, con una complejidad de O(n), lo que se refleja en tiempos de ejecución menores en comparación con otros métodos.

seleccionInversaMultiple_1 es un poco menos eficiente debido a la necesidad de realizar múltiples recorridos sobre la lista y marcar elementos, con una complejidad de O (3n/2).

seleccionInversaMultiple_2 es sorprendentemente eficiente debido a la rapidez a la hora de recorrer un array en sentido inverso y eliminar elementos dándonos una complejidad de O(n/2)

particion es el método menos eficiente en términos de tiempo de ejecución, con una complejidad de $O(n^2)$ en el peor de los casos, lo que hace que su rendimiento se degrade significativamente a medida que el tamaño de la lista aumenta.

Los tiempos de ejecución medidos en las pruebas siguen las tendencias esperadas según las complejidades teóricas calculadas, confirmando que el análisis matemático previo es una herramienta válida para predecir el rendimiento de los algoritmos.

Además, gracias a este trabajo observamos la importancia de una correcta implementación para obtener una correcta optimización y rendimiento del código que desarrollamos, ya que, al trabajar con grandes volúmenes de datos, las diferencias en el tiempo de ejecución entre distintas implementaciones pueden ser significativas y repercutir en la fluidez de los programas creados.