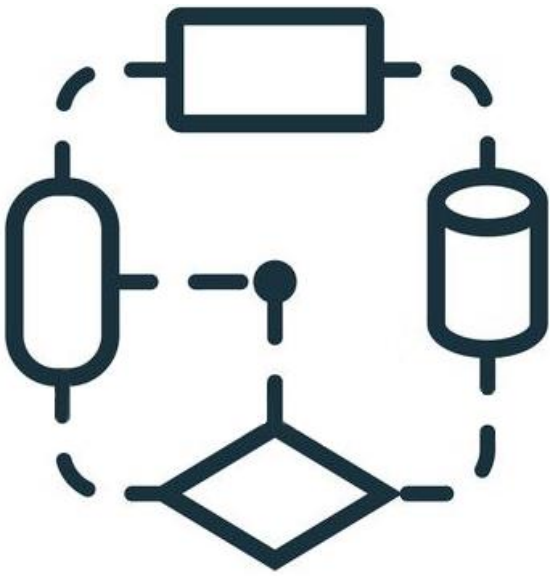
Greedy
(esquema voraz)

PRÁCTICA 2



Autor: Pablo Gómez Rivas y Jesús Fornieles Muñoz

Materia: Estructura de Datos y Algoritmos II

Grupo de Prácticas: GTAB2

Fecha: Domingo, 23 de abril de 2023

Índice

Objetivo

El problema que se presenta en el texto es conocido como el problema de la mochila, que es un problema de optimización combinatoria. En este problema, se tiene una mochila con una capacidad limitada y un conjunto de objetos, cada uno con su propio peso y valor. El objetivo es maximizar el valor total de los objetos que se pueden llevar en la mochila sin exceder su capacidad máxima.

En este caso, Jero tiene que decidir qué objetos puede llevarse del tesoro que ha descubierto, teniendo en cuenta su peso y el límite de capacidad de su mochila de espeleólogo. Para resolver este problema, puede utilizar un algoritmo greedy que seleccione los objetos con mayor valor por unidad de peso y los incluya en la mochila hasta que se alcance su capacidad máxima. Este enfoque greedy puede proporcionar una solución subóptima pero rápida para el problema de la mochila.

Como hemos mencionado, en esta práctica se ha utilizado un algoritmo de tipo Greedy para resolver el problema de la mochila presentado en el texto anterior. El objetivo de este apartado es realizar un estudio teórico de este método algorítmico y su aplicación en este caso particular.

El algoritmo Greedy es un método heurístico que busca solucionar un problema de optimización a través de una estrategia de selección de elementos de forma voraz, es decir, en cada paso se escoge la opción que aparentemente parece la mejor en ese momento, sin considerar posibles consecuencias futuras. En otras palabras, el algoritmo Greedy siempre escoge la opción localmente óptima en cada paso, con la esperanza de que esto lleve a una solución globalmente óptima. De esta forma, determinamos nuestro mecanismo de elección local óptima a la ordenación de la lista sobre la que vamos a operar en función del valor o valor/peso de cada uno de los objetos que Jero pueda meter en su mochila, de forma que siempre empezamos eligiendo aquellos valores que nos pueden otorgar un mayor valor, finalizando la ejecución con una solución óptima que no necesariamente tiene que ser la mejor:

objetos.sort(Comparator.comparingDouble(Objeto::getValorUnitario).reversed());

Resulta importante mencionar que el algoritmo Greedy no siempre lleva a la solución óptima y puede encontrar soluciones subóptimas en algunos casos. Por lo tanto, es necesario tener en cuenta que su aplicación debe ser evaluada en función del problema específico que se está abordando y de sus características particulares.

En resumen, el algoritmo Greedy es una técnica heurística eficiente que puede proporcionar soluciones aproximadas en un tiempo razonable para problemas de optimización, como el problema de la mochila presentado en esta práctica. Sin embargo, su aplicación debe ser cuidadosamente evaluada en función de las características del problema y se deben considerar otras estrategias de resolución en caso de que el algoritmo no proporcione los resultados deseados.

Por otro lado, en lo que respecta a este trabajo, hemos decidido hacer a Pablo Gómez Rivas líder de esta práctica, encargado de supervisar y verificar la calidad y el correcto funcionamiento de los códigos del trabajo del resto de miembros, Jesús Fornieles Muñoz en este caso, de forma que la elaboración del proyecto ha quedado de la siguiente forma:

#	Tipo	Tarea	
1	Implementacion	Algortimo voraz MochilaFraccionada	Pablo
2	Implementacion	Algortimo voraz MochilaEntera	Jesús
3	Implementacion	Algortimo voraz MochilaMitad	Pablo
4	Implementacion	Implementación de Lingotes	Jesús
5	Implementacion	Implementación de Lingotes infinitos	Pablo
6	Implementacion	Generación de secuencuas aleatorias. Considerar casos especiales	Jesús
7	Implementacion	Main con medidas de tiempos de ejecución	Pablo
8	Implementacion	Juegos de prueba, considerando los casos especiales	Jesús
9	Documentación	**Estudio de la implementación**	Pablo
10	Documentación	**Estudio teórico**	Jesús
11	Documentación	**Estudio experimental**	Pablo
12	Documentación	**Anexos** y bibliografia	Jesús

Antecedentes

El club de espeleología de EDAland está en pleno apogeo y cada vez más aventureros se inscriben para saciar su sed de aventura. Hoy nuestro protagonista es Jeronimo, más conocido como Jero el Aventurero, el cual ha decidido explorar el peligroso pico Zornoko (el segundo pico más alto de todo Almeriroc).

Una vez iniciada su expedición, Jero celebra que ha sido todo un éxito pues ha encontrado una gran cantidad de variados tesoros, pero a la hora de elegir cuales se llevará consigo se encuentra con una premisa, ¿Cuál es la combinación de objetos de mayor valor que puede cargar en su mochila?

Para darle solución a esta pregunta surgen varias formas de contestar, y no dependen de Jero sino del tipo de mochila que cargue consigo, en la cual encontramos 3 variantes:

Mochila 0-1:

Para este tipo de mochila Jero podrá almacenar dentro de su mochila cualquier cantidad (entera o decimal) de cualquier tesoro, siempre que esa cantidad no rebase la capacidad de la mochila. De esta forma Jero tiende a llenar su mochila con aquellos tesoros con una mayor ratio valor/peso hasta dejarla completa.

Mochila 0/1:

Para este nuevo tipo de mochila Jero tendrá que decidir si carga o no el objeto entero, pues este no se puede fraccionar. Para ello Jero buscará la combinación de objetos que quepan en la mochila y que hagan que en su conjunto tenga el máximo valor posible.

Mochila 0/0,5/1:

En este tipo de mochila Jero podrá almacenar nada, la mitad o el tesoro completo en su mochila. Esta metodología resulta de una pequeña fusión de las dos mochilas anteriores debido a que tendrá valores fijos para la elección de cada tesoro en función de su peso, como ocurre en la mochila 0/1, pero a su vez seguirá un orden en función del valor como lo hacía la mochila 0-1.

Estudio de la Implementación

Mochila 0-1:

Este algoritmo cuenta con una complejidad $O(n \log n)$ ocasionada por el método sort que ordena la lista de objetos.

La base de este consiste en un algoritmo greedy, solo que, dada las características de la mochila, siempre nos va a ofrecer una solución óptima. Para realizar la selección de los objetos hacemos uso del método Sort de la clase List, al cual le añadimos como parámetro el atributo valor unitario (ratio valor/peso) para que ordene nuestra lista de objetos en función del atributo anterior en orden descendente. Una vez con la lista ordenada procedemos a introducir en la mochila (mediante el for) los elementos de mayor unitario hasta que llegamos al punto en el que el siguiente objeto tiene un peso mayor que la capacidad restante, en este punto planteamos dos caminos:

La mochila ya está llena (capacidad actual = 0): no introducimos ninguna fracción de ningún objeto posterior.

La mochila no está llena (capacidad actual > 0): introducimos en la mochila una cantidad fraccionada del objeto tal que objeto.peso =

capacidad actual, siendo entonces capacidad actual * objeto.valor, el valor de la fracción del objeto que añadimos a la mochila.

Observando así que, dada la ordenación inicial según el valor unitario, el algoritmo siempre ofrece la mejor solución posible.

Mochila 0/1:

Para este nuevo planteamiento del problema de la mochila encontramos una variación en la que solo podemos meter o bien el objeto entero o bien no meterlo.

Para ello vamos a realizar una técnica ciertamente parecida a la que hemos aplicado en el algoritmo anterior (de nuevo orden de complejidad O(n log n)), es decir, ordenamos los objetos por su valor unitario para después proceder a introducirlos en función de dicho atributo.

La variación que supone este algoritmo es que ya no podemos fraccionar los objetos para introducirlos en la mochila, por lo que si al iterar los objetos nos encontramos un objeto cuyo peso sobrepase la capacidad actual de la mochila lo ignoramos, de forma que seguimos explorando la lista de objetos hasta encontrar aquellos que si quepan según nuestra capacidad actual.

De esta forma es cierto que no vamos a conseguir la configuración óptima de la mochila para obtener el valor máximo posible según los objetos de forma muy frecuente, pero al ser un algoritmo greedy nos ofrece una solución bastante aceptable acompañado de un orden de complejidad reducido, orden de complejidad O(n log n) en este caso.

Mochila 0/0,5/1:

Para esta nueva versión, observamos que podemos introducir en la mochila los objetos enteros, la mitad o no introducirlos. Para ello planteamos un algoritmo muy similar al anterior, pero con una pequeña variación a la hora de seleccionar qué objetos pueden entrar en la mochila. Para aplicar esta variación nos centramos en el momento en el que iteramos los objetos, donde una cada vez que el objeto no entra completo en la mochila no lo descartamos y probamos con el siguiente, sino que tratamos de probar a introducir en la mochila la mitad del objeto, es decir: si objeto.peso > capacidad actual probamos si objeto.peso/2 > capacidad actual, de forma que en caso de que la mitad del peso del objeto no sobrepase la capacidad actual este será añadido con la mitad de peso y la mitad de valor, para seguir así probando con los siguientes elementos de la lista de objetos y con sus respectivas mitades en su defecto.

De nuevo nos encontramos con un algoritmo greedy que no va a devolver la mejor solución frecuentemente, pero que nos ofrece una buena solución con un orden de complejidad reducido, en este caso de orden n.

Principales diferencias entre problema de la mochila 0/1 y el problema de la mochila 0-1:

A continuación, se muestra una tabla comparativa que resume las principales diferencias entre ambas variantes del problema desde un punto de vista greedy:

	Problema de la mochila 0/1	**Problema de la mochila 0-1 (objetos fraccionables)**
Greedy	Se utiliza un algoritmo greedy para seleccionar los objetos que se van a incluir en la mochila. En cada iteración, se selecciona el objeto con el mayor valor, siempre y cuando quepa en la mochila.	Se utiliza un algoritmo greedy para seleccionar los objetos que se van a incluir en la mochila. En cada iteración, se selecciona una fracción del objeto que proporcione el mayor valor por unidad de peso, siempre y cuando quepa en la mochila.
Condición de optimalidad	La solución óptima se alcanza seleccionando siempre el objeto con mayor valor que quepa en la mochila, hasta que no haya más objetos que quepan en ella.	La solución óptima se alcanza seleccionando fracciones de los objetos con mayor valor por unidad de peso, hasta que la mochila esté llena.
Complejidad temporal	El algoritmo greedy tiene una complejidad temporal de O(n log n), donde n es el número de objetos.	El algoritmo greedy tiene una complejidad temporal de O(n log n), donde n es el número de objetos.

Limitaciones	El algoritmo greedy no garantiza encontrar la solución óptima en todos los casos, especialmente si los objetos tienen pesos y valores muy dispares.	El algoritmo greedy garantiza encontrar la solución óptima siempre y cuando se puedan seleccionar fracciones de los objetos. Si no es posible seleccionar fracciones de los objetos, el problema de la mochila 0/1 es el adecuado.

En resumen, el algoritmo greedy utilizado en el problema de la mochila 0/1 puede no ser óptimo en todos los casos, mientras que el algoritmo greedy utilizado en el problema de la mochila 0-1 siempre encuentra la solución óptima.

Estudio Teórico

Los tres métodos presentados tienen el mismo orden de complejidad ***O*(n log n)** porque comparten la misma estructura básica de algoritmo. Todos ellos comienzan ordenando una lista de objetos en orden descendente según su valor unitario, lo que significa que esta operación de ordenamiento ya consume ***O*(n log n)** tiempo. Luego, se inicializan la capacidad actual y la lista de objetos seleccionados, y se itera sobre la lista de objetos seleccionando elementos según la capacidad disponible en ese momento. Aunque las condiciones de selección de objetos varían entre los métodos, el proceso general de iterar sobre la lista ordenada y seleccionar objetos según la capacidad disponible es el mismo, lo que implica que tienen el mismo orden de complejidad.

Dada esta similitud, vamos a presentar solamente el estudio teórico del algoritmo Mochila Fraccionaria, por ejemplo:

```
public List<Objeto> getMochilaFraccionaria() {
          List<Objeto> objetos = new
ArrayList<>(this.objetos);

     objetos.sort(Comparator.comparingDouble(Objeto::getVal
orUnitario).reversed());
          double capacidadActual = capacidad;
          List<Objeto> sol = new ArrayList<>();
          for (Objeto objeto : objetos) {
               if (objeto.getPeso() <= capacidadActual) {
                    capacidadActual -= objeto.getPeso();
                    sol.add(objeto);
```

```
                    if (capacidadActual == 0)
                        break;
                } else {
                    sol.add(new Objeto(capacidadActual,
capacidadActual * objeto.getValorUnitario()));
                    break;
                }
            }
            return sol;
        }
```

- El orden de complejidad de las líneas no marcadas es $O(1)$ ya que son operaciones elementales de tiempo constante.
- El orden de complejidad de la ordenación de la lista, `objetos.sort(Comparator.comparingDouble(Objeto::getValorUnitario).reversed());`, es $O(\text{n log n})$.
- El orden de complejidad del bucle `for (int j = i; j < arr.length; j++)` es $O(\text{n})$ ($\sum_{i=1}^{n} d$).

  se calcula:

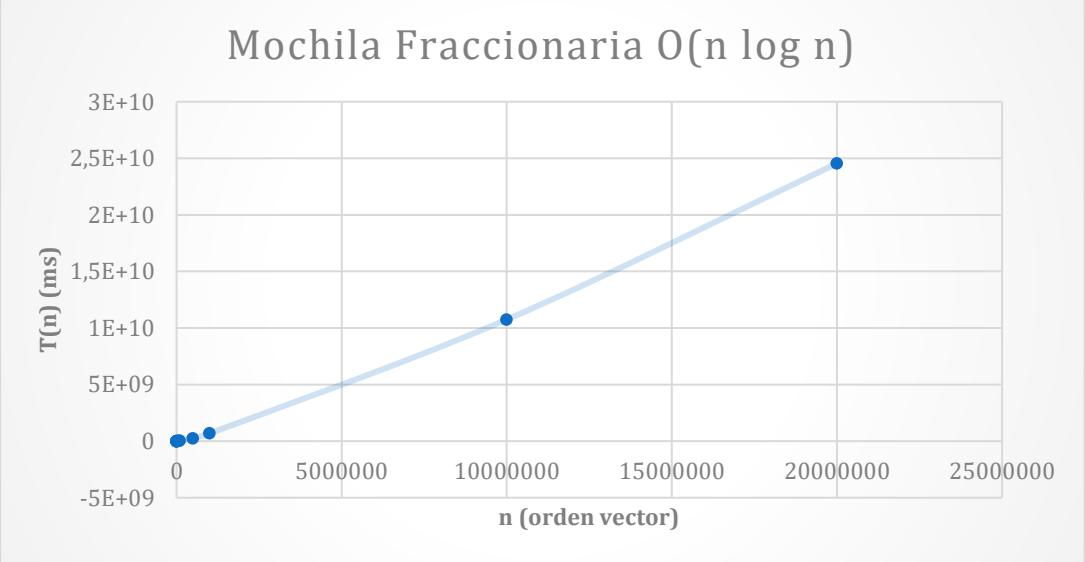
$$\bar{T}(\text{n}) \leq O(n \log n) + O(n) \in \boldsymbol{O}(\mathbf{n \log n})$$

Estudio Experimental

Algoritmo Mochila Fraccionaria O(n log n)

n (tamaño lista)	$\bar{T}(n)$ (ms)
50	99395
200	279030
1000	483330
5000	2989540
50000	1,92E+07
100000	3,81E+07
500000	2,53E+08
1000000	6,82E+08
10000000	1,08E+10
20000000	2,45E+10

Tiempo de Ejecución VS Tamaño de la Entrada

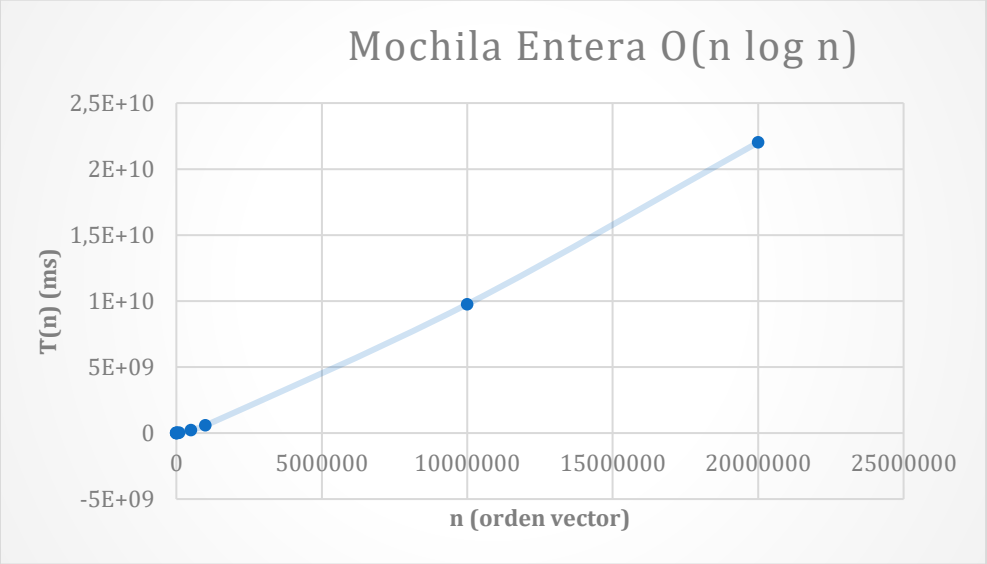


Gráfica tiempo de ejecución VS Tamaño de la entrada

Algoritmo Mochila Entera O(n log n)

n (tamaño lista)	$\bar{T}(n)$ (ms)
50	76255
200	337295
1000	1021850
5000	3114690
50000	13.035.750
100000	26.379.815
500000	236.509.235
1000000	577.278.545
10000000	9.759.733.070
20000000	22.016.727.270

Tiempo de Ejecución VS Tamaño de la Entrada

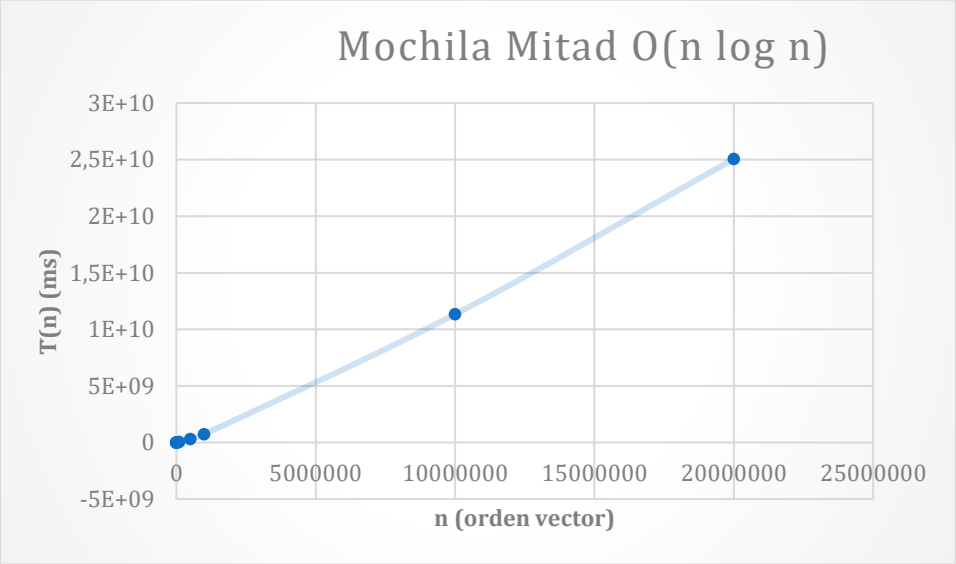


Gráfica tiempo de ejecución VS Tamaño de la entrada

Algoritmo Mochila Mitad O(n log n)

n (tamaño lista)	$\bar{T}(n)$ (ms)
50	68265
200	188105
1000	748070
5000	2319665
50000	17471665
100000	37400775
500000	299243780
1000000	733239420
10000000	11331118500
20000000	25060786370

Tiempo de Ejecución VS Tamaño de la Entrada

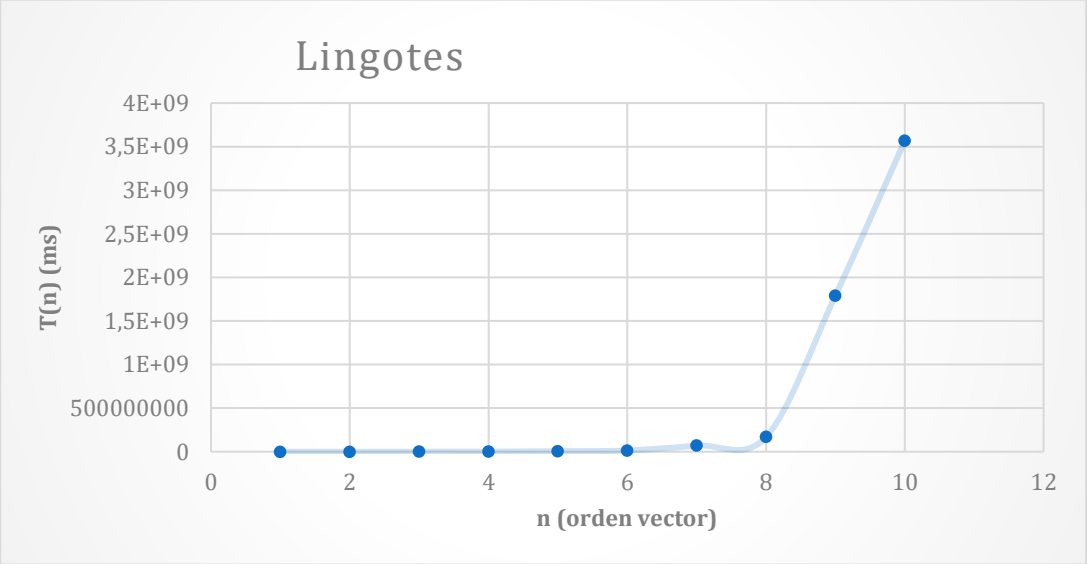


Gráfica tiempo de ejecución VS Tamaño de la entrada

Algoritmo Lingotes O(n log n)

n (tamaño lista)	$\bar{T}(n)$ (ms)
50	67365
200	110810
1000	138405
5000	946480
50000	6041470
100000	14289220
500000	70521605
1000000	169563260
10000000	1792421710
20000000	3566572470

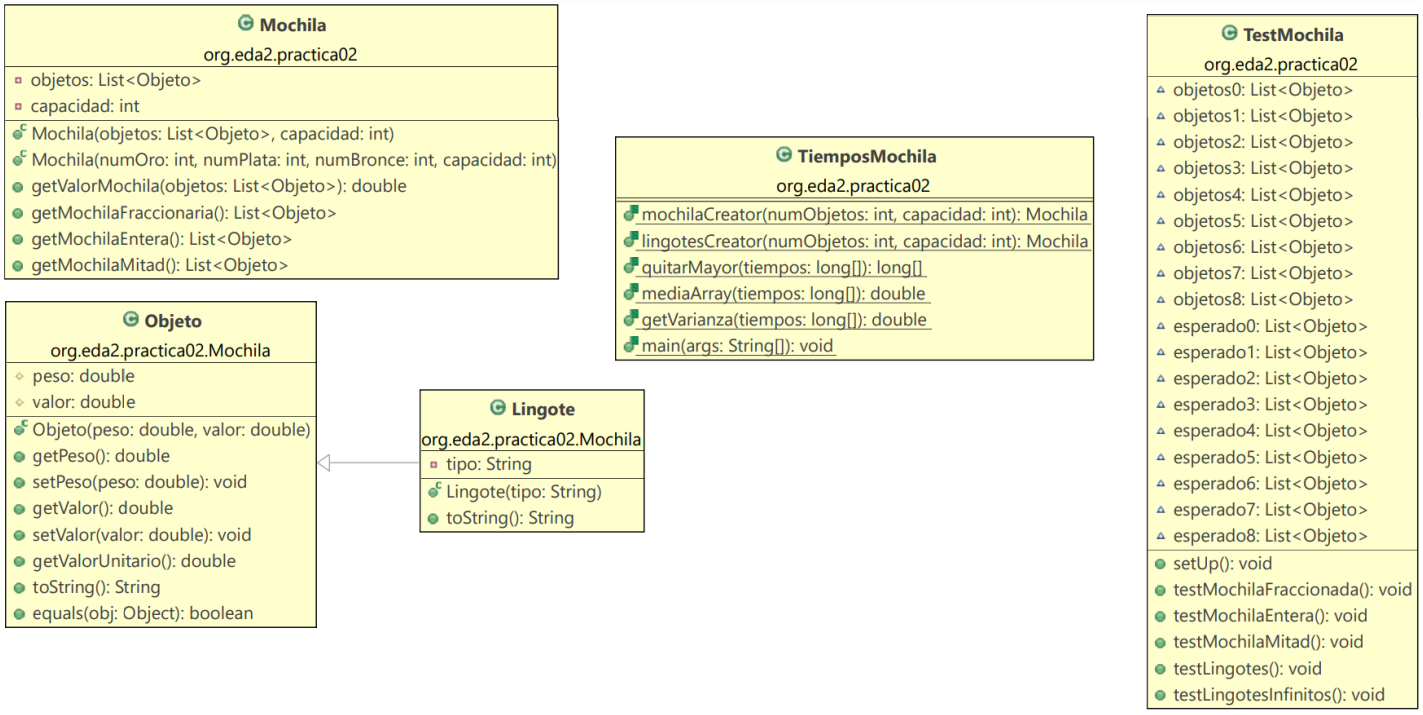
Tiempo de Ejecución VS Tamaño de la Entrada



Gráfica tiempo de ejecución VS Tamaño de la entrada

Anexo

Diagrama de clases:



Archivos fuente:

- **Mochila:**

En este archivo dispuesto como clase java podremos encontrar los 3 métodos exigidos durante la práctica destinados a la orientación y gestión de los tipos de mochila, junto a una serie de métodos (getters) y clases auxiliares (objeto y lingote).

- **TiemposMochila:**

En este archivo dispuesto como clase java podemos encontrar una amigable interfaz que nos permite comprobar el tiempo de cualquiera de los algoritmos de la práctica con el n(tamaño de vector de entrada) que deseemos agregar.

- **TestMochila:**

En este archivo dispuesto como clase java observamos la existencia de una serie de diversos test Junit 5 que nos permiten probar nuestro código con una variedad de objetos generados de forma aleatoria cuyo valor escala hasta 10 millones para probar así nuestro código de la forma más completa posible.

Fuentes Bibliográficas

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, **"Introduction to Algorithms"** 3rd ed. MIT Press, 2009.

2. S. Sridhar, **"Algorithms for Interviews"** CreateSpace Independent Publishing Platform, 2014.

3. R. Lafore, **"Data Structures and Algorithms in Java"** 2nd ed. Sams Publishing, 2002.

4. M. T. Goodrich and R. Tamassia, **"Data Structures and Algorithms in Java"** 6th ed. John Wiley & Sons, 2021.

5. G. Brassard and P. Bratley, **"Fundamentals of Algorithmics"** 2nd ed. Prentice Hall, 1996.

6. V. Antoniali, **"Dynamic Programming and Optimal Control"** Springer, 2020.

7. C. M. Bishop, **"Pattern Recognition and Machine Learning"** Springer, 2006.