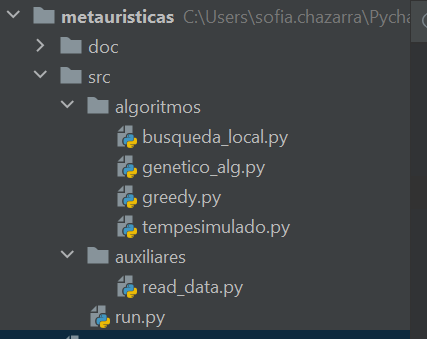
OPTIMIZACIÓN II

# 1. Descripción del código.

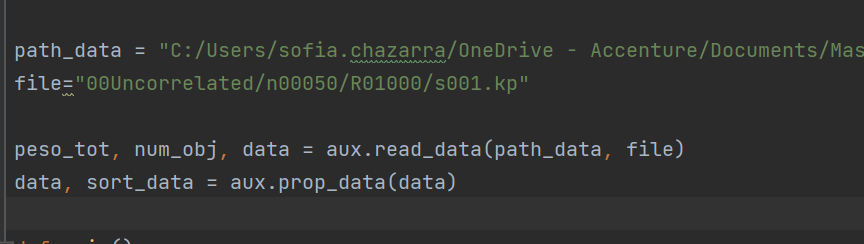
El código está desarrollado con el lenguaje Python versión 3.7. Dentro del archivo enviado, tenemos la siguiente composición de carpetas:



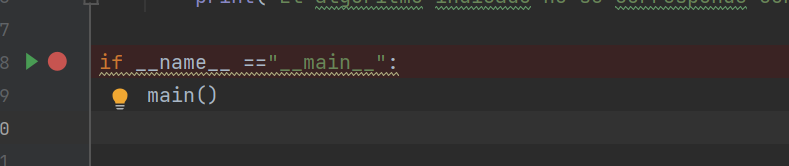
La documentación necesaria para entender el código, así como de las pruebas realizadas con los distintos algoritmos, la encontraremos en **doc.**  La carpeta de **src** contiene tanto los algoritmos principales, como las funciones auxiliares que permiten el correcto funcionamiento de los algoritmos. Dentro de la carpeta **algoritmos**, vemos que encontramos un script de Python para todos los algoritmos que hemos desarrollado que describiremos en el siguiente punto. Por otra parte, en la carpeta **auxiliares**, tenemos un único script read\_data.py donde hay funciones como:

1. **read\_data**: que se encarga de la correcta lectura de los archivos que se le pasan como argumento.
2. **Prop\_data**: calcula la división del peso y la importancia, para darle un porcentaje de “valor” a ese elemento.
3. **Calcular\_peso**: dada una solución y la población inicial seleccionada, se calcula el peso de la mochila.
4. **Calcula\_valor**: dada una solución y la población inicial seleccionada, se calcula el valor de la mochila.

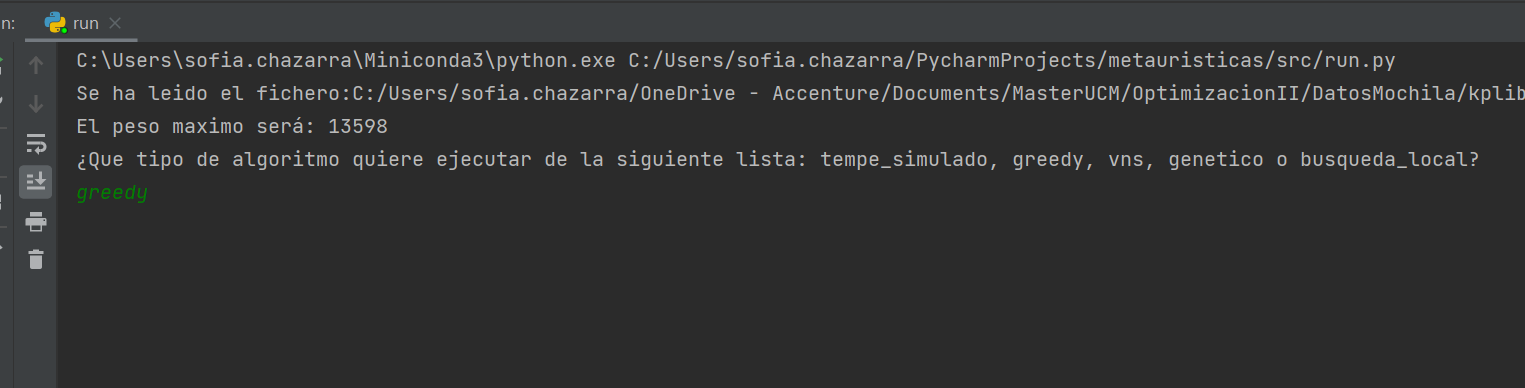
Por último, tenemos el script run.py, que contiene nuestra función principal que ejecutaremos para cualquiera de todos nuestros algoritmos. En la primera parte del script definimos la ruta general del archivo que vamos a ejecutar (path\_data), y el nombre del archivo y sus carpetas específicas(file). A partir de esto, se ejecuta la función read\_data que nos devuelve los datos, así como el peso máximo y el numero de objetos. Tras esto, ejecutamos prop\_data, cuya salida la utilizaremos para el algoritmo greedy.



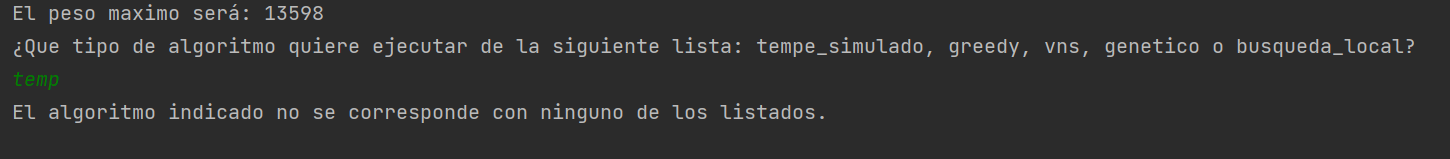
La función principal o main, se ejecuta pulsando el botón de play:



Y está nos pedirá que indiquemos el algoritmo a ejecutar dada una lista de nombres. Deberemos escribir por terminal el nombre del algoritmo que queremos ejecutar (greedy en el siguiente ejemplo):



Nos devolverá un mensaje de error si no encuentra el algoritmo seleccionado:

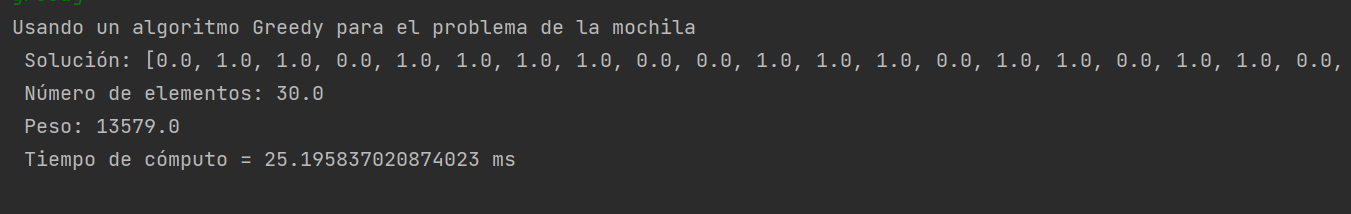


# 2. Algoritmos

## Greedy

El algoritmo constructivo escogido ha sido el greedy. Para este, hemos calculado con la función de prop\_data la relación entre peso y valor, para considerar los elementos con mayor valor y menor peso mediante esta proporción. Así, para construir una solución inicial seleccionamos los objetos con menor proporción hasta que llegue al peso máximo establecido por los datos de entrada.

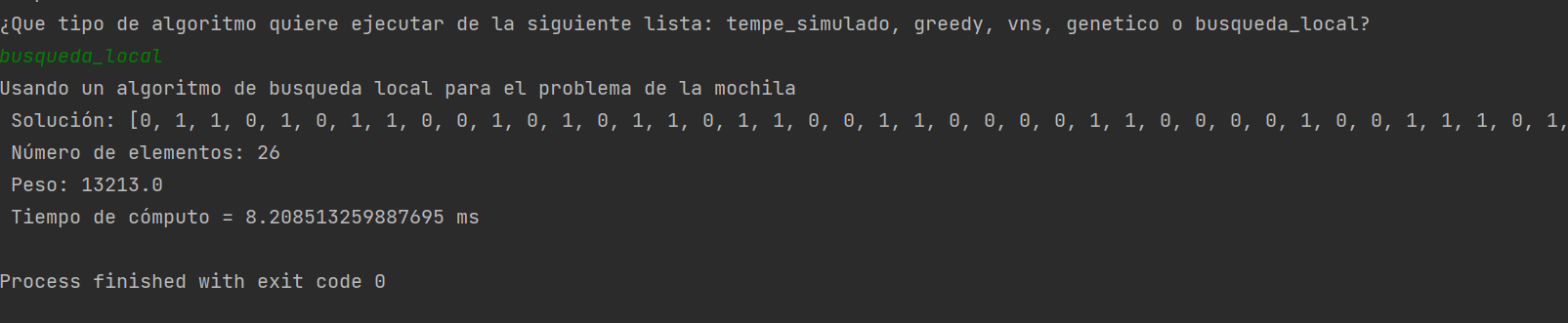
La salida del algoritmo nos dará la solución, que será una lista de 0’s y 1’s dependiendo si elegimos o no el objeto para meterlo en la mochila. También nos señala el número de objetos que se depositan en la mochila, el peso acumulado de dichos objetos y tiempo total de cómputo del algoritmo.



## Búsqueda local

El algoritmo voraz escogido ha sido la búsqueda\_local. Para este, creamos una solución inicial de 0’s y 1’s a partir del algoritmo greedy, y creamos 3 tipos de entornos (variable entorno = 0,1, u 2) para valorar distintos tipos de casos de sacar y meter objetos en la mochila. Así pues, para el entorno 0, será el caso en el que haya que sacar dos objetos de la mochila y cambiarlos por 1. .Para el entorno 1, estaremos en el caso en el que haya que sacar un elemento de la mochila y cambiarlo por otro.Y para el entorno 2, simplemente vamos a añadir más elementos a la mochila.

La salida del algoritmo nos dará la solución, que será una lista de 0’s y 1’s dependiendo si elegimos o no el objeto para meterlo en la mochila. También nos señala el número de objetos que se depositan en la mochila, el peso acumulado de dichos objetos y tiempo total de cómputo del algoritmo.



## Tempe simulado

La idea del temple simulado es ir mejorando una solución inicial a medida que se disminuye la “temperatura”. Esta solución inicial nos la proporciona el algoritmo greedy y la temperatura inicial será un parámetro a calibrar.

El algoritmo consiste en realizar sucesivas búsquedas locales para ir actualizando la solución. Trataremos de mejorar la solución inicial en la medida de lo posible hasta que se cumplan los criterios de parada. Estos criterios de parada tendrán relación con la temperatura así como con el número de iteraciones del algoritmo.

La salida del algoritmo nos dará la solución, que será una lista de 0’s y 1’s dependiendo si elegimos o no el objeto para meterlo en la mochila. También nos señala el número de objetos que se depositan en la mochila y tiempo total de cómputo del algoritmo.

## VNS

La idea de la Búsqueda en Entorno Variable es realizar sucesivas búsquedas locales explorando los distintos entornos posibles de las soluciones.

Para nuestra búsqueda entorno variable hemos decidido implementar una Búsqueda en Entorno Variable General. La Búsqueda en Entorno Variable General se apoya en la Búsqueda Descendente, que consiste en realizar búsquedas locales sobre los distintos entornos de manera sistemática.

Se comienza con una solución inicial proporcionada por el algoritmo greedy. La idea es escoger aleatoriamente una estructura de entornos y realizar una búsqueda local sobre ella. A continuación, se realiza una Búsqueda Descendente a partir de esta solución. Este proceso se repite hasta que se alcancen los criterios de parada.

## Genético

Para el Algoritmo genético vamos a definir en primer lugar, una clase para abordar el problema mediante un algoritmo genético general. Los atributos de los que constan son:

* genes: lista de posibles valores [0,1] de genes en un cromosoma.
* decodifica: lista inicial de cromosomas.
* longitud\_individuos: nº de cromosomas.
* iteraciones: número de iteraciones que se hacen sobre los cromosomas.

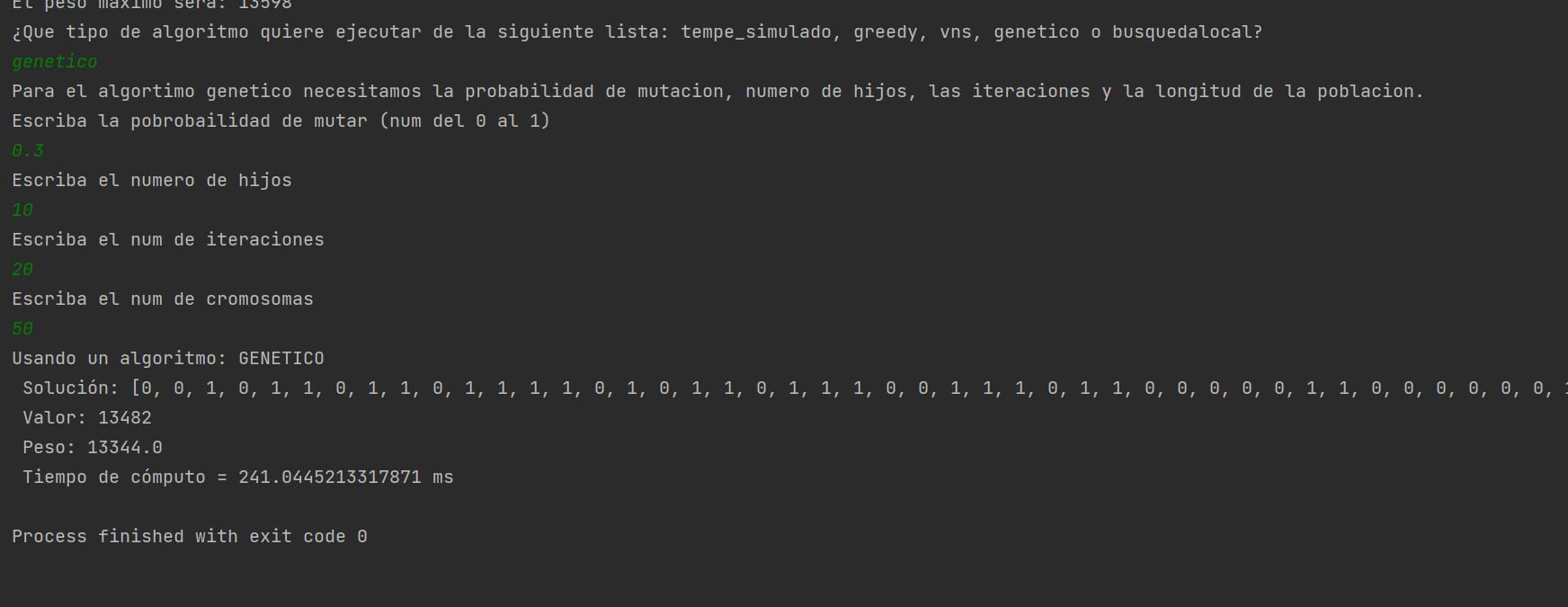
Definimos las funciones de raíz de mutación y cruce, y la más importante: la función objetivo, que utilizamos para elegir los genes dentro de la población dada. Esta función crea una lista de listas [valor, solución], en la que el valor indica el valor total de la mochila (Si el peso es inferior al peso\_maximo), o el peso máximo menos el peso de la solución (si supera el valor máximo, por lo que dicho valor será negativo). Ordenamos de mayor a menor según los valores, y cogemos las soluciones de la población con valores más altos. Es decir, cogemos las mejores soluciones sin que sobrepasen el peso máximo de la mochila.

La función mutación, se encarga en cambiar valores aleatorios de ceros y unos de las soluciones en función de una probabilidad dada (prob\_mutar), y estudia con la función objetivo los nuevos mejores valores de dichas soluciones mutadas.

La función cruce recibe dos parámetros de entrada, que será la población1 y la población2, o lo que es lo mismo, los padres. Esta función crea una nueva población con las combinaciones aleatorias de los dos padres, dando lugar a los hijos. Para escoger a los padres, creamos una función ruleta que se encargue de escoger aleatoriamente a x participantes que crearán la poblacion1 y poblacion2. Se eliminarán de la antigua población los participantes, escogidos y se añadirán los hijos (que tendrán la longitud que hayamos escogido nosotros al llamar la función con el argumento sol\_rep).

Así pues, dentro de la función main llamada “genético”, creamos una lista con nº: poblacion\_size de poblaciones iniciales. Cada población inicial tiene una longitud de num\_obj, y le aplicamos nuestra función objetivo. Tras esto, instanciamos nuestra clase objetivo de Problema\_genetico, y a partir de ahí, y en base al número de iteraciones que hayamos definido, hacemos que compitan las soluciones mediante los cruces, y tras esta nueva población, le aplicamos las funciones de mutación.

Para ejecutarlo, pedirá 4 parámetros de entrada: la probabilidad de mutar (que será un número del 0 al 1), el número de hijos (que tendrá que ser menor que la longitud de la solución), el numero de iteraciones y el nº de cromosomas que confronta nuestra población.

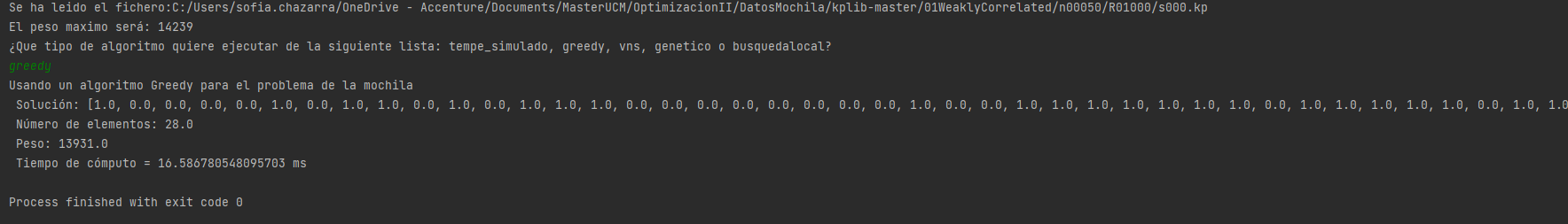


# 3. Pruebas realizadas

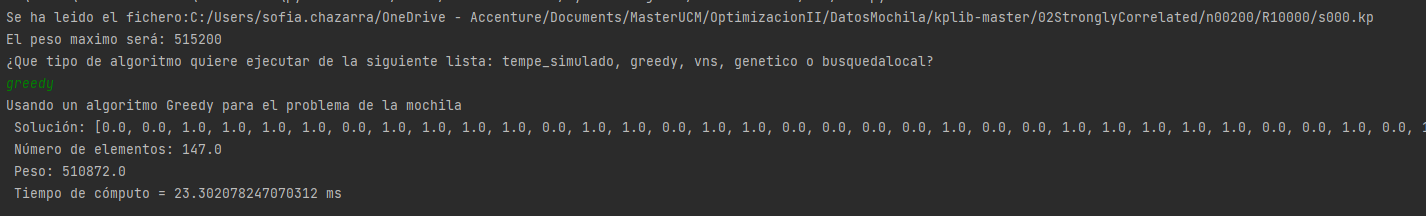
## 3.1 Greedy

Hemos probado con 3 tipos de instancia y hemos obtenido los siguientes resultados:

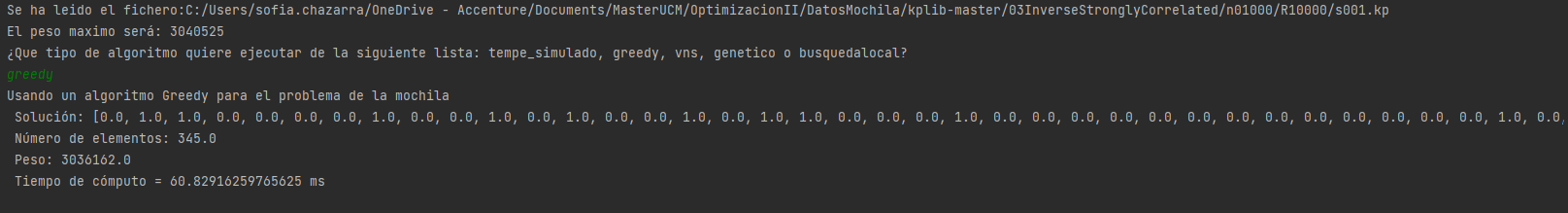
1. 01WeaklyCorrelated\n00050\R01000\s000.kp



1. 02StronglyCorrelated\n00200\R10000\s000.kp

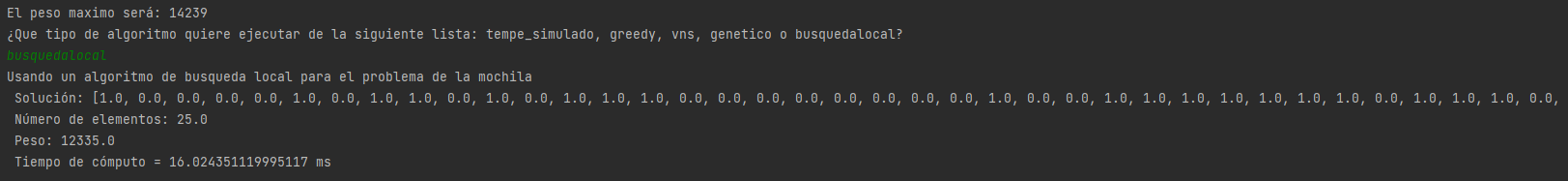


1. 03InverseStronglyCorrelated\n01000\R10000\s001.kp

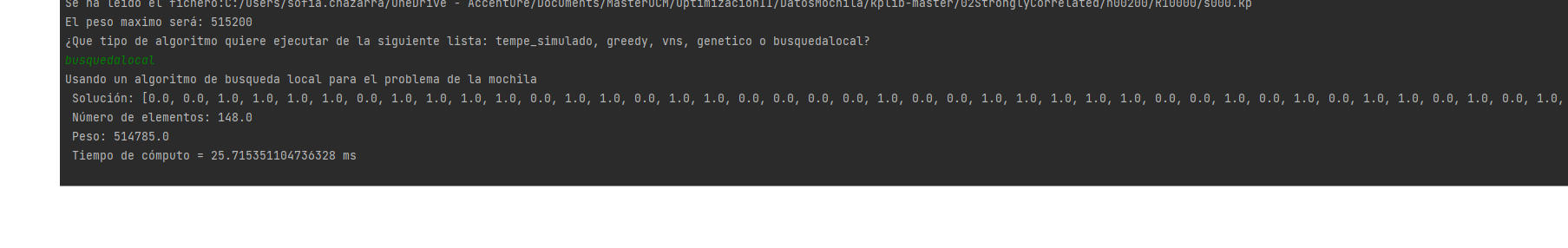


## Busqueda local

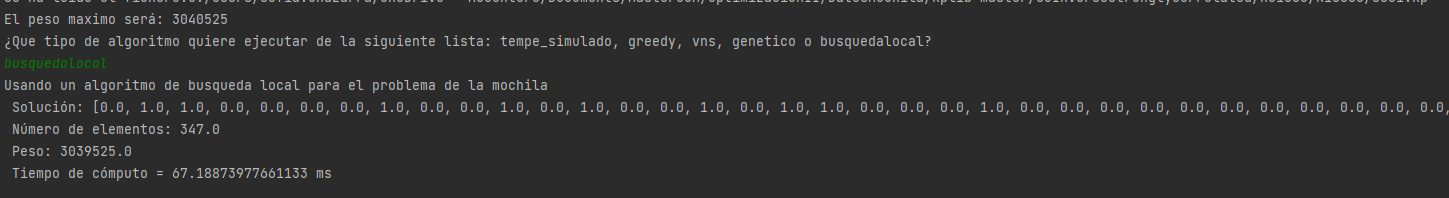
1. 01WeaklyCorrelated\n00050\R01000\s000.kp



1. 02StronglyCorrelated\n00200\R10000\s000.kp

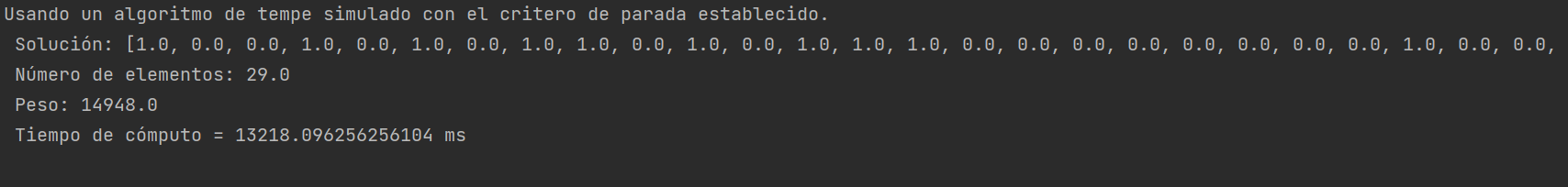


1. 03InverseStronglyCorrelated\n01000\R10000\s001.kp



## 3.3 Tempe simulado

Para el temple simulado, y con un criterio de parada aleatorio para t en un rango del o al 200, tenemos que:



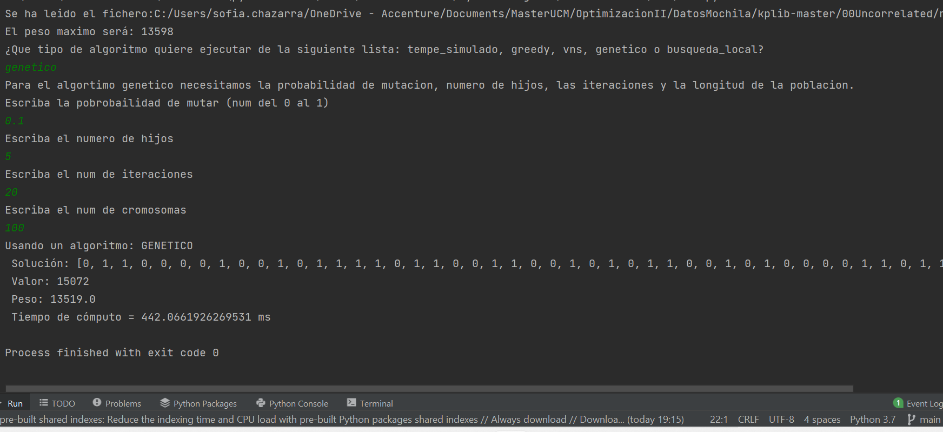
Para los datos del ejemplo 01WeaklyCorrelated\n00050\R01000\s000.kp

## Genético

Para las pruebas con el genético he utilizado la instancia de “00Uncorrelated/n00050/R01000/s001.kp”. El mejor valor obtenido ha sido con los parámetros de entrada de:

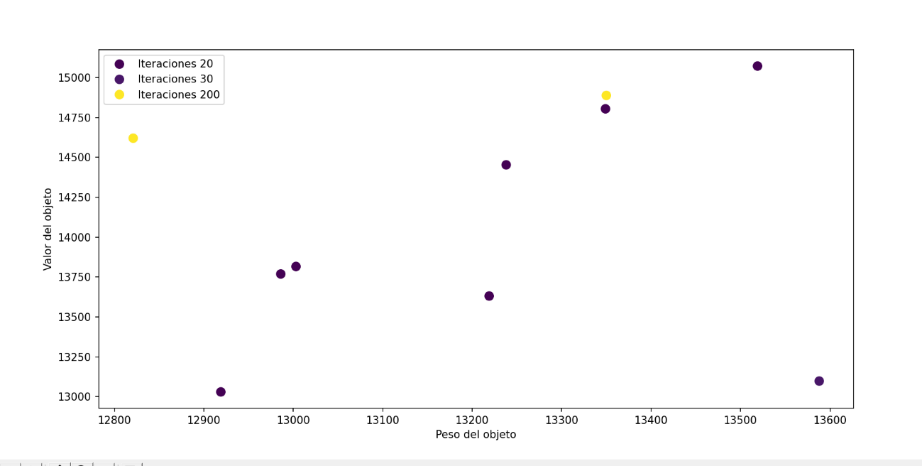
* Probabilidad de mutar: 0.1
* Número de hijos: 5
* Iteraciones: 20
* Cromosomas: 100

El valor obtenido será 15072 y el peso 13519.

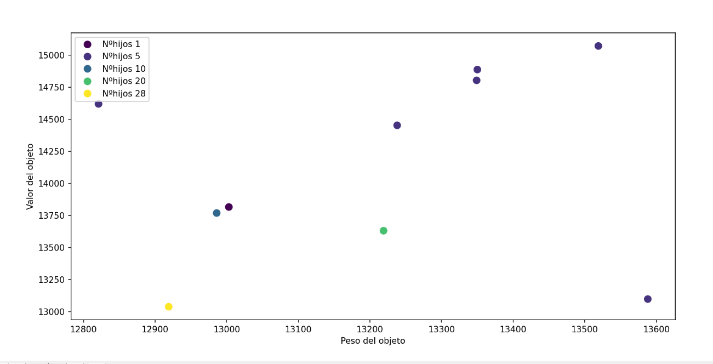


He obtenido las siguientes gráficas para representar como varia la relación entre valor y peso en función de los parámetros que hayamos seleccionado.

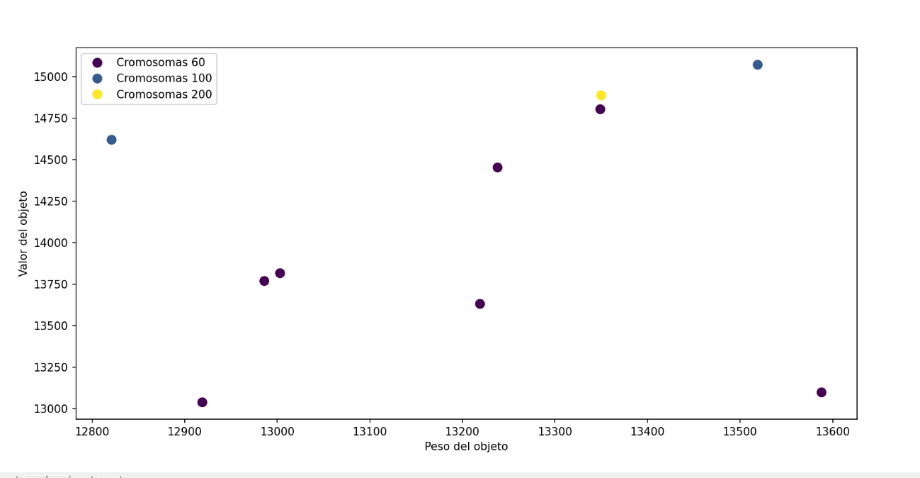
Para este caso, aumentando las iteraciones no aumenta la relación valor-objeto. De hecho, el mejor valor se obtiene con 20 iteraciones.



Cambiando el número de hijos, vemos que se obtienen mejores valores para los casos de 1 y 5 hijos, que para los casos de 10, 20 y 28.



En los cromosomas no se puede sacar una relación clara entre el número escogido y los valores representados. Dependerán también de los otros parámetros escogidos.



Probando con los casos 0.1 y 0.4 para la probabilidad de mutar, vemos como el calor amarillo (prob 0.4) representa valor-peso bajos, mientras que el 0.1 representa valor-peso más altos.

