Pruebas de Software 2023-24

PEC: Prueba de algoritmos que resuelven el Problema de la Mochila

Dpto. de Ingeniería de Software y Sistemas Informáticos

Rubén Heradio, rheradio@issi.uned.es

ETSI Informática, Universidad Nacional de Educación a Distancia

Índice

1. Enunciado

El **problema de la mochila**[1], comúnmente abreviado como **KP** (del inglés *Knapsack Problem*), es un problema de optimización que busca la mejor solución entre un conjunto, generalmente enorme, de soluciones posibles. El KP consiste en llenar una mochila que puede soportar hasta un peso determinado, con todo o parte de un conjunto de artículos, cada uno con un peso y valor específicos. Los objetos colocados en la mochila deben maximizar el valor total sin exceder el peso máximo.

La Figura 1 muestra un ejemplo de mochila con un capacidad máxima de 8 kg. De entre los artículos 0-4 disponibles, debemos escoger aquellos que sumen un valor máximo, pero con un peso total que no exceda la capacidad de la mochila. En este caso, la solución óptima consiste en seleccionar los artículos 2-4, cuyo peso es $1 + 3 + 4 = 8$ kg y valor $1 + 10 + 5 = 16$ €.

Figura 1: Instancia del KP

Este problema tiene aplicaciones prácticas en áreas como la economía y la ingeniería, donde a menudo se persigue maximizar un beneficio sin exceder cierto límite establecido. Además, el KP es conocido por su complejidad computacional, siendo uno de los 21 problemas NP-completos de R. Karp[2]. Por lo que es un área de estudio relevante en teoría de la computación e investigación operativa.

[1] Ver https://en.wikipedia.org/wiki/Knapsack_problem para una descripción detallada
[2] https://en.wikipedia.org/wiki/Karp%27s_21_NP-complete_problems

Esta práctica consiste en probar varios algoritmos, escritos en `Python`[3,4], que resuelven el KP.

- Las Secciones 1.1-1.6 describen el material proporcionado para realizar la práctica.
- **Las Secciones 1.7 y 1.8 enuncian las tareas obligatorias** que deben entregarse para aprobar la práctica. Con la realización de estas tareas podrá obtenerse, como máximo, una calificación de 8 en esta práctica.
- **La Sección 1.9 propone una tarea voluntaria** cuya realización permite alcanzar la calificación de 10.

> **Aviso**:
> Cada estudiante debe resolver la PEC individualmente. El equipo docente utilizará herramientas de detección de plagio para inspeccionar el código y la memoria de la PEC. La copia no está permitida y conllevará el suspenso de la asignatura.

1.1. Código proporcionado para realizar la práctica

Puede obtener el código asociado a esta práctica en el curso virtual de la asignatura (para acceder a dicho material, debe autentificarse en Campus UNED[5]):

`https://agora.uned.es/mod/resource/view.php?id=221946`

Como puede apreciarse en el siguiente diagrama, este código se organiza en dos carpetas llamadas `mochila` y `tests`.

```
pec
├── mochila
│   ├── mochila.py ..................... Código que hay que probar
│   ├── main.py ........................ Programa de ejemplo
│   ├── busqueda_exhaustiva.py ......... Código que hay que probar
│   ├── busqueda_con_poda.py ........... Código que hay que probar
│   └── algoritmo_voraz.py ............. Código que hay que probar
└── tests ............................... Carpeta donde deben colocarse los tests
    ├── __init__.py .................... ¡No tocar este fichero!
    └── test_ejemplo.py ................ Ejemplo de test
```

La práctica consistirá en probar los ficheros `mochila.py`, `busqueda_exhaustiva.py`, `busqueda_con_poda.py` y `algoritmo_voraz.py`, cuyo contenido se describe en las Secciones 1.2-1.5.

[3] El intérprete de Python puede descargarse en `https://www.python.org/downloads/`
[4] Puede consultarse un tutorial sobre Python en `https://docs.python.org/es/3/tutorial/`
[5] `https://www.uned.es/`

La carpeta `tests` incluye el fichero `test_ejemplo.py`, que se describe en la Sección 1.6. Esta carpeta incluye además el fichero `__init__.py`, que está vacío y no debe borrarse, porque facilitará la ejecución de `pytest`[6]

1.2. Estructuras de datos

El Listado 1 muestra las estructuras de datos que emplearemos. Una `Mochila` (Líneas 13-57) puede almacenar varios `Artículos` (Líneas 1-10), cuyo peso no puede sumar más de la `capacidad` de la mochila.

```
1  class Articulo:
2    def __init__(self, valor, peso):
3      self.valor = valor
4      self.peso = peso
5      self.seleccionado = False
6
7    def __str__(self):
8      return f"valor = {self.valor}, " \
9          f"peso = {self.peso}, " \
10         f"seleccionado = {self.seleccionado}"
11
12
13 class Mochila:
14   def __init__(self, capacidad=0):
15     self.articulos = []
16     self.capacidad = capacidad
17
18   def insertar_articulo(self, valor, peso):
19     art = Articulo(valor, peso)
20     self.articulos.append(art)
21
22   def suma_valores(self, sumar_todos=False):
23     suma = 0
24     for art in self.articulos:
25       if art.seleccionado or sumar_todos:
26         suma += art.valor
27     return suma
28
29   def suma_pesos(self, sumar_todos=False):
30     suma = 0
31     for art in self.articulos:
32       if art.seleccionado or sumar_todos:
33         suma += art.peso
34     return suma
35
36   def valor(self):
37     if self.suma_pesos() > self.capacidad:
38       return -1
39     else:
40       return self.suma_valores()
41
42   def articulo_de_max_valor(self, peso_disponible):
43     max_valor = 0
44     i = -1
45     for j in range(len(self.articulos)):
46       if ((not self.articulos[j].seleccionado)
47           and (self.articulos[j].valor > max_valor)
```

[6] Puede encontrar una explicación sobre la conveniencia de `__init__.py` en el Apéndice 4 de [1].

```
48            and (self.articulos[j].peso <= peso_disponible)):
49          max_valor = self.articulos[j].valor
50          i = j
51      return i
52
53  def imprimir(self, imprimir_todos=True):
54      for i in range(len(self.articulos)):
55        if self.articulos[i].seleccionado or imprimir_todos:
56          print("articulo ", i, ": ", self.articulos[i])
57      print("Maximo peso permitido: ", self.capacidad, "\n")
```

Listado 1: Estructuras de datos para resolver el KP (`mochila.py`)

La clase `Artículo` incluye los siguientes métodos:

1. El constructor `__init__` (Líneas 2-5), que inicializa el peso y valor de un artículo. Además, inicializa un *flag de selección* a `False`, para indicar que el artículo aún no se ha elegido para incluirse en la mochila.

2. `__str__` (Líneas 7-10) devuelve una cadena de caracteres para facilitar la impresión del problema (Líneas 53-57).

La clase `Mochila` incluye los siguientes métodos:

1. El constructor `__init__` (Líneas 14-16), que inicializa la capacidad y la lista de artículos disponibles. El parámetro `capacidad` es opcional y por defecto vale 0 kg. La capacidad de una mochila nunca debería ser un número negativo de kilos.

2. `insertar_articulo` (Líneas 18-20) crea un objeto de la clase `Artículo` y lo añade a la lista de artículos disponibles. El peso de un artículo siempre debe ser > 0 y su valor ≥ 0.

3. `suma_valores` (Líneas 22-27) devuelve la suma total de los valores de los artículos. El parámetro opcional `sumar_todos`, que por defecto es `False`, indica si los artículos no seleccionados deben considerarse en la suma.

4. `suma_pesos` (Líneas 29-34) es análogo a `suma_valores`, pero para pesos. Devuelve la suma total de los pesos de los artículos. El parámetro opcional `sumar_todos`, que por defecto es `False`, indica si los artículos no seleccionados deben considerarse en la suma.

5. `valor` (Líneas 36-40) devuelve el valor obtenido por una solución del KP. Es decir, `suma_valores()` si `suma_pesos` $\leq$ `capacidad`, ó -1 en caso contrario.

6. `articulo_de_max_valor` (Líneas 42-51) devuelve el índice del artículo con mayor valor de todos los no seleccionados en lista de artículos disponibles, siempre y cuando su peso sea $\leq$ `peso_disponible`. Si ningún artículo cumple la restricción de `peso_disponible`, devuelve -1.

7. `imprimir` (Líneas 53-57) visualiza la estructura de datos imprimiendo la capacidad de la mochila, la lista de elementos disponibles y qué artículos se han seleccionado para resolver el problema.

Los Listados 2 y 3 muestran cómo codificar el ejemplo de la Figura 1 usando las clases `Artículo` y `Mochila`.

```
1 from mochila import Mochila
2
3 mochila = Mochila(8)
4 mochila.insertar_articulo(10, 6)
5 mochila.insertar_articulo(2, 2)
6 mochila.insertar_articulo(1, 1)
7 mochila.insertar_articulo(10, 3)
8 mochila.insertar_articulo(5, 4)
9 mochila.imprimir()
```

Listado 2: Ejemplo de programa que utiliza la clase `Mochila` (`main.py`)

```
1 articulo 0 : valor = 10, peso = 6, seleccionado = False
2 articulo 1 : valor = 2, peso = 2, seleccionado = False
3 articulo 2 : valor = 1, peso = 1, seleccionado = False
4 articulo 3 : valor = 10, peso = 3, seleccionado = False
5 articulo 4 : valor = 5, peso = 4, seleccionado = False
6 Maximo peso permitido: 8
```

Listado 3: Resultado en consola de ejecutar el programa del Listado 2

1.3. Algoritmo 1: búsqueda exhaustiva

El KP se puede resolver como un *problema de decisión*. Es decir, como una secuencia de decisiones que conducen a una solución. En nuestro caso, las decisiones son si cada artículo debe incluirse en la mochila. Cada posible solución es una asignación completa de "síes" y "noes" a todos los artículos. Nuestro objetivo es encontrar la solución legal (cuyos artículos caben en la mochila) que tenga el máximo valor total posible. Debemos tener en cuenta que la "mejor" solución puede no ser única, es decir, habrá casos en los que varias selecciones distintas de artículos sumen el mismo valor total óptimo. En esos casos, el KP debería devolver cualquiera de esas soluciones óptimas.

Los problemas de decisión se suelen representar como un árbol binario como el que se muestra en la Figura 2. Los nodos en un nivel i del árbol representan el i-ésimo artículo. Las ramas que salen de un nodo representan la decisión de incluir el artículo i en la mochila o no. Supongamos que la rama izquierda significa que el artículo i se incluye y la rama derecha que no.

Cada camino desde el nodo raíz hasta una hoja en la parte inferior del árbol representa una solución completa. Siguiendo una estrategia de *fuerza bruta*, podemos examinar todas las posibles soluciones siguiendo todos los caminos desde la raíz hasta las hojas, y quedándonos con la que solución que alcance mayor valor.

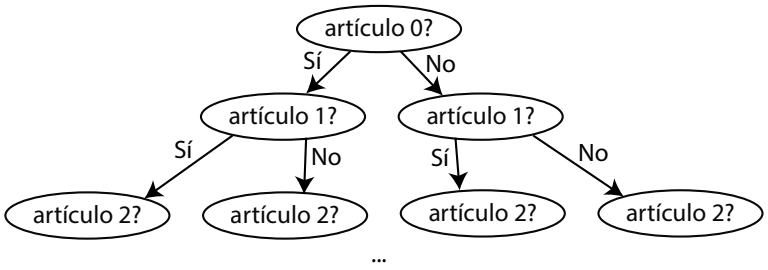

Figura 2: Árbol de decisión binario para el KP

El Listado 4 implementa esta estrategia de forma recursiva.

```
1  from copy import deepcopy
2
3  def busqueda_exhaustiva(mochila):
4    return haz_busqueda_exhaustiva(mochila, 0)
5
6  def haz_busqueda_exhaustiva(mochila, indice_articulo):
7    if indice_articulo == len(mochila.articulos):
8      return (deepcopy(mochila), mochila.valor())
9    else:
10     mochila.articulos[indice_articulo].seleccionado = True
11     solucion_sel_art, valor_sel_art \
12       = haz_busqueda_exhaustiva(mochila, indice_articulo + 1)
13     mochila.articulos[indice_articulo].seleccionado = False
14     solucion_no_sel_art, valor_no_sel_art \
15       = haz_busqueda_exhaustiva(mochila, indice_articulo + 1)
16     if valor_sel_art > valor_no_sel_art:
17       return (solucion_sel_art, valor_sel_art)
18     else:
19       return (solucion_no_sel_art, valor_no_sel_art)
```

Listado 4: Resolución del KP mediante búsqueda exhaustiva (`busqueda_exhaustiva.py`)

Empezando desde la raíz o nivel 0 (Línea 4), cada llamada recursiva avanza progresivamente un nivel más en el árbol. En concreto:

1. Las Líneas 11-12 nos llevan por la rama izquierda desde nuestra posición actual, explorando la posibilidad de que el artículo i se incluya en la mochila por medio de una llamada recursiva a `haz_busqueda_exhaustiva`.

2. Cuando la función recursiva regresa, las Líneas 14-15 exploran el caso contrario, es decir, que el artículo i no se incluya en la mochila (rama derecha).

3. Cuando la segunda llamada recursiva finaliza, las variables `valor_sel_art` y `valor_no_sel_art` contienen el valor alcanzado por las dos ramas exploradas. Las Líneas 16-19 comparan dichos valores y se quedan con la mejor solución.

4. Las Líneas 7-8 implementan el caso base. La recursión finaliza cuando se alcanzan una hoja del árbol. En este punto, se calcula el valor de la solución llamando al método `valor` definido en la Líneas 36-40 del Listado 1.

En `Python`, los objetos se pasan por referencia. Como no queremos modificar el valor del argumento `mochila`, se devuelve una copia de la solución (Línea 8: `deepcopy(mochila)`).

1.4. Algoritmo 2: búsqueda con poda

El algoritmo de búsqueda exhaustiva es muy ineficiente. Si hay n artículos disponibles, necesita realizar 2^n llamadas recursivas. Sin embargo, muchas de esas llamadas son inútiles y se pueden evitar.

El Listado 5 muestra una versión mejorada, que va guardando en la variable `mejor_valor_hasta_ahora` el máximo valor logrado por las soluciones examinadas (Líneas 23-24) y evita llamadas recursivas cuando:

1. El valor de la solución que estamos explorando no puede mejorar el `mejor_valor_hasta_ahora`. Es decir, cuando el valor de la solución parcial actual más el valor de todos los artículos que quedarían por explorar es inferior a la `mejor_valor_hasta_ahora` (Líneas 29-30).

2. La solución actual viola las restricciones de peso. Es decir, cuando el peso de la solución parcial actual supera la capacidad de la mochila (Líneas 34-35).

```
1  from copy import deepcopy
2
3  def busqueda_con_poda(mochila):
4    indice_articulo = 0
5    valor_actual = 0
6    peso_actual = 0
7    valor_restante = mochila.suma_valores(True)
8    mejor_valor_hasta_ahora = [0]
9    return haz_busqueda_con_poda(mochila, indice_articulo,
10                   valor_actual,
11                   peso_actual,
12                   valor_restante,
13                   mejor_valor_hasta_ahora)
14
15
16 def haz_busqueda_con_poda(mochila, indice_articulo,
17               valor_actual,
18               peso_actual,
19               valor_restante,
20               mejor_valor_hasta_ahora):
21   if indice_articulo == len(mochila.articulos):
22     valor = mochila.valor()
23     if valor > mejor_valor_hasta_ahora[0]:
24       mejor_valor_hasta_ahora[0] = valor
25     return (deepcopy(mochila), mochila.valor())
26
27   else:
28
29     if valor_actual + valor_restante <= mejor_valor_hasta_ahora[0]:
30       return (None, 0)
31
32     solucion_sel_art = None
33     valor_sel_art = 0
```

```
34    if peso_actual + mochila.articulos[indice_articulo].peso \
35        <= mochila.capacidad:
36      mochila.articulos[indice_articulo].seleccionado = True
37      solucion_sel_art, valor_sel_art \
38        = haz_busqueda_con_poda(mochila, indice_articulo + 1,
39                    valor_actual + mochila.articulos[indice_articulo].valor,
40                    peso_actual + mochila.articulos[indice_articulo].peso,
41                    valor_restante - mochila.articulos[indice_articulo].valor,
42                    mejor_valor_hasta_ahora)
43
44    mochila.articulos[indice_articulo].seleccionado = False
45    solucion_no_sel_art, valor_no_sel_art \
46      = haz_busqueda_con_poda(mochila, indice_articulo + 1,
47                  valor_actual,
48                  peso_actual,
49                  valor_restante - mochila.articulos[indice_articulo].valor,
50                  mejor_valor_hasta_ahora)
51
52    if valor_sel_art > valor_no_sel_art:
53      return (solucion_sel_art, valor_sel_art)
54    else:
55      return (solucion_no_sel_art, valor_no_sel_art)
```

Listado 5: Resolución del KP mediante búsqueda con poda (`busqueda_con_poda.py`)

1.5. Algoritmo 3: estrategia voraz

El Listado 6 resuelve el KP de una forma más rudimentaria que los algoritmos descritos en las Secciones 1.3 y 1.4 anteriores. Simplemente, escoge iterativamente el artículo con mayor valor que quepa en mochila.

```
1  from copy import deepcopy
2
3  def algoritmo_voraz(mochila):
4    solucion = deepcopy(mochila)
5    peso_disponible = solucion.capacidad
6    valor = 0
7    while peso_disponible > 0:
8      i = solucion.articulo_de_max_valor(peso_disponible)
9      if i == -1:
10       break
11     solucion.articulos[i].seleccionado = True
12     valor += solucion.articulos[i].valor
13     peso_disponible -= solucion.articulos[i].peso
14   return solucion, valor
```

Listado 6: Resolución del KP mediante un algortimo voraz (`algoritmo_voraz.py`)

1.6. Ejemplo de test

El Listado 7 muestra un juego de pruebas de ejemplo, escrito con la librería `pytest` [1], para el algoritmo de búsqueda exhaustiva descrito en la Sección 1.3. Consulte las PECs de 2022[7] y 2023[8], disponibles en el curso virtual, para aprender

[7] https://agora.uned.es/mod/resource/view.php?id=119965
[8] https://agora.uned.es/mod/resource/view.php?id=214689

cómo instalar `pytest`, así como su funcionamiento básico.

```python
import pytest
from mochila.mochila import Mochila
from mochila.busqueda_exhaustiva import busqueda_exhaustiva


def test_ejemplo():
  mochila = Mochila(8)
  mochila.insertar_articulo(10, 6)
  mochila.insertar_articulo(2, 2)
  mochila.insertar_articulo(1, 1)
  mochila.insertar_articulo(10, 3)
  mochila.insertar_articulo(5, 4)
  solucion, valor = busqueda_exhaustiva(mochila)
  assert valor == 16
  assert solucion.articulos[0].seleccionado == False
  assert solucion.articulos[1].seleccionado == False
  assert solucion.articulos[2].seleccionado == True
  assert solucion.articulos[3].seleccionado == True
  assert solucion.articulos[4].seleccionado == True
```

Listado 7: Ejemplo de test para `busqueda_exhaustiva.py` (`test_ejemplo.py`)

Para ejecutar el test, sitúese en la carpeta `pec` usando el terminal de su sistema operativo y ejecute el comando:

```
pytest
```

1.7. Tarea 1: Probar `busqueda_exhaustiva`

1.7.1. Clases de equivalencia y valores límite

Utilice la metodología explicada en los *Ejemplos de PECs 1*[9] *y 2*[10], disponibles en el curso virtual, para:

1. **Identificar las dimensiones de interés** para los parámetros de entrada de `busqueda_exhaustiva` y su salida.
2. **Identificar las clases de equivalencia, valores límite y valores de prueba** para cada una de esas dimensiones. **Deberá resumir dicha información en una tabla análoga a la Tabla 1 del** *Ejemplo de PEC 1*.
3. **Identificar las restricciones** que deben darse entre esas dimensiones de interés para que las pruebas "tengan sentido".

En la corrección de su práctica, se tendrá en cuenta que describa detalladamente la metodología que ha seguido, así como que justifique razonadamente los valores de prueba escogidos y sus restricciones.

[9] https://agora.uned.es/mod/resource/view.php?id=119952
[10] https://agora.uned.es/mod/resource/view.php?id=119953

1.7.2. Especificación ACTS de los valores de prueba

Obtenga con ACTS dos Juegos de Pruebas (JPs) alternativos para $t = 2$:

1. **JP1**: en el primero, la especificación ACTS considerará los valores válidos y no válidos de todas las variables.

2. **JP2**: en el segundo, se excluirán las variables que puedan tener valores inválidos. Si una variable adquiere uno de esos valores, debería abortarse la ejecución del programa y lanzarse algún tipo de excepción (p. ej., si la mochila tuviera una capacidad negativa).

¿En qué se diferencian los dos juegos de pruebas? ¿Qué implicaciones prácticas tienen dichas diferencias?

1.7.3. Pruebas de unidades con pytest

Con el fin de automatizar las pruebas, **codifique con la librería `pytest` [1] los tests que implementan el juego de pruebas más conveniente.** Es decir, uno de los dos, JP1 ó JP2.

En caso de elegir JP2, añada tests que comprueben qué sucede cuando las variables excluidas tienen valores inválidos.

Debe **escribir todos estos tests en un fichero llamado `test_busqueda_exhaustiva.py` dentro de la carpeta `tests`**.

1.7.4. Corrección del programa

En caso de que sus pruebas detecten algún error, **realice las correcciones oportunas en los Listados 1 y 4**.

1.8. Tarea 2: Probar busqueda_con_poda y algoritmo_voraz

1.8.1. Testing diferencial

Pruebe `busqueda_con_poda` y `algoritmo_voraz`. Para ello, **siga la estrategia conocida como** *testing diferencial*. Esta estrategia se puede aplicar cuando ya existe una implementación alternativa validada del programa que se desea probar. En nuestro caso, ése programa alternativo, que utilizaremos como *oráculo*, será `busqueda_exhaustiva`.

Deberá **escribir un fichero llamado `test_algoritmos_alternativos.py`, dentro de la carpeta `tests`**, con el siguiente contenido:

1. Una función auxiliar `genera_aleatorio`, que generará una instancia aleatoria del KP con un nº de artículos que se especificará como parámetro de entrada. La capacidad de la mochila resultante será la mitad de la suma total de los pesos de los artículos generados (utilice la división entera). Para generar números

aleatorios en `Python`, puede utilizar la función `randint(desde, hasta)` de la libraría `random` (la importación sería `from random import randint`). Por ejemplo, `randint(0, 10)` genera un entero entre 0 y 10.

2. Dos pruebas, `test_busqueda_con_poda` y `test_algoritmo_voraz`, que llamarán 100 veces a `genera_aleatorio`, para producir 100 problemas KP de diversos tamaños (entre 5 y 10 artículos), y comprobarán si el valor de sus soluciones coincide con el conseguido con `busqueda_exhaustiva`.

A la vista de los resultados, ¿son correctos `busqueda_con_poda` y `algoritmo_voraz`?

1.8.2. Escalabilidad de los algoritmos

Añada a `test_busqueda_exhaustiva.py` dos funciones adicionales que generen dos ficheros CSV[11] (de *Valores Separados por Comas*):

1. `test_escalabilidad_exhaustiva_vs_poda` (i) generará instancias aleatorias de KP con $5, 6, 7, \ldots, 17$ artículos disponibles; (ii) ejecutará sobre dichas instancias `busqueda_exhaustiva` y `busqueda_con_poda`; y (iii) guardará en un fichero CSV, llamado `escalabilidad_exhaustiva_vs_poda.csv`, los siguientes campos: `algoritmo` utilizado, `numero_de_articulos` de la instancia del KP, `valor_optimo` de la solución obtenida y `segundos` necesarios para ejecutar el algoritmo.

   El Listado 8 muestra un fragmento de ejemplo del CSV que debe obtener.

```
algoritmo,numero_de_articulos,valor_optimo,segundos
busqueda_exhaustiva,5,238,0.0006277561187744141
busqueda_con_poda,5,238,0.0
busqueda_exhaustiva,6,198,0.002159595489501953
busqueda_con_poda,6,198,0.0
...
busqueda_exhaustiva,17,560,11.0468590259552
busqueda_con_poda,17,560,0.0009965896606445312
```

Listado 8: Ejemplo de fichero `escalabilidad_exhaustiva_vs_poda.csv`

2. `test_escalabilidad_poda_vs_voraz` (i) generará instancias aleatorias de KP con $5, 6, 7, \ldots, 34$ artículos disponibles; (ii) ejecutará sobre dichas instancias `busqueda_con_poda` y `algoritmo_voraz`; y (iii) guardará en un fichero CSV, llamado `escalabilidad_poda_vs_voraz.csv`, los mismos campos que el punto anterior: algoritmo, numero_de_articulos, valor_optimo y segundos.

[11] https://es.wikipedia.org/wiki/Valores_separados_por_comas

Para calcular el tiempo de ejecución de cada algoritmo, utilice la función `time` de la librería `time`, que obtiene la hora de su ordenador. Por ejemplo, el Listado 9 muestra cómo obtener los segundos que han transcurrido en una ejecución de `busqueda_exhaustiva`.

```
from time import time
...
hora_antes = time()
solucion_busq_exhaustiva, valor_busq_exhaustiva = busqueda_exhaustiva(mochila)
hora_despues = time()
segundos = hora_despues - hora_antes
```

Listado 9: Ejemplo de cálculo del tiempo de ejecución con `time`

Para generar los ficheros CSV, utilice la función `writerow` de la librería `csv`. Utilice el Listado 10 como ejemplo.

```
import csv
...
writer = csv.writer(file)
writer.writerow(["algoritmo", "numero_de_articulos", "valor_optimo", "segundos"])
for numero_de_articulos in range(5, 18):
...
  writer.writerow(["busqueda_exhaustiva", numero_de_articulos, valor_busq_exhaustiva, segundos])
```

Listado 10: Ejemplo de escriitura de un fichero CSV

Analice gráficamente los ficheros CSV generados para contestar a las siguientes preguntas:

1. ¿Qué algoritmo es más rápido? ¿Cómo varía la velocidad de los algoritmos a medida que aumentan los artículos?
2. En caso de que el algoritmo voraz no sea 100 % correcto, ¿se acercan sus soluciones a las obtenidas por `busqueda_exhaustiva` y `busqueda_con_poda`?

Para el análisis gráfico, puede emplear la herramienta que desee. Por ejemplo:

- Microsoft Excel: https://support.microsoft.com/es-es/office/crear-un-gr%C3%A1fico-de-principio-a-fin-0baf399e-dd61-4e18-8a73-b3fd5d5680c2
- Google Sheets: https://youtu.be/BABC1jktDIc

1.9. Tarea 3: Implementación y prueba de una solución basada en programación dinámica (opcional)

El problema de la mochila puede resolverse eficientemente mediante la técnica de *programación dinámica*. Por ejemplo, consulte:

1. https://www.youtube.com/watch?v=IZHvQTx2bZ0
2. https://en.wikipedia.org/wiki/Knapsack_problem#0-1_knapsack_problem

Implemente el KP utilizando programación dinámica y la estructura de datos del Listado 1. Compruebe su corrección y escalabilidad utilizando el método explicado en la Tarea 2. Compare los tiempos de ejecución del algoritmo basado en programación dinámica con el algoritmo voraz.

Recuerde que esta tarea es voluntaria y le de acceso a la máxima puntuación en la práctica (un 10).

2. Material que debe entregarse en el curso virtual

En el curso virtual, deberá entregar un fichero comprimido titulado `PrimerApellido_SegundoApellido_Nombre.zip` (Por ejemplo, `Garcia_Gil_Andrea`). El fichero contendrá:

1. Un `pdf` con la memoria de su solución.
2. Las especificaciones `ACTS` descritas en la Sección 1.7.2.
3. Los juegos de prueba generados por `ACTS`.
4. Fichero `test_busqueda_exhaustiva.py` definido en la Sección 1.7.3
5. El código `Python` de la versión corregida de los Listados 1 y 4.
6. Fichero `test_algoritmos_alternativos.py` definido en la Sección 1.8.1
7. Ficheros CSV definidos en la Sección 1.8.2
8. **Opcional:** (i) Fichero `programacion_dinamica.py` que resuelve el KP utilizando programación dinámica (ver Sección 1.9); (ii) Fichero `test_programacion_dinamica.py` que valida dicho algoritmo con testing diferencial y genera un CSV con tiempos de ejecución de programación dinámica *vs* algoritmo voraz, (iii) Fichero CSV generado, y (iv) análisis gráfico de los datos en el CSV.

Referencias

[1] Brian Okken. *Python Testing with pytest: Simple, Rapid, Effective, and Scalable*. The Pragmatic Programmers, 2017.