TRABAJO PRÁCTICO N°2 - PROBLEMA DE LA MOCHILA

CICLO LECTIVO 2025

Mercé Alexis
Universidad Tecnológica Nacional
Ingeniería en Sistemas
Legajo 50174
alexis.am.2001@gmail.com

Neirotti Bruno
Universidad Tecnológica Nacional
Ingeniería en Sistemas
Legajo 47792
bneirotti@gmail.com

Sebben Mateo
Universidad Tecnológica Nacional
Ingeniería en Sistemas
Legajo 49609
mateosebben114@gmail.com

9 de agosto de 2025

RESUMEN

Este trabajo aborda el problema de la mochila, un caso clásico de optimización combinatoria, mediante la implementación y comparación de dos enfoques: la Búsqueda Exhaustiva y el Algoritmo Greedy. A su vez, se desarrolla y explica un programa ejecutable para resolver instancias con diferentes restricciones (volumen y peso), analizando las particularidades, ventajas y limitaciones de cada método en términos de eficiencia y calidad de las soluciones. El estudio permite evaluar cómo la elección del algoritmo impacta en el rendimiento y en la adecuación de la solución frente a distintas escalas y condiciones del problema.

Índice

1. Introducción

El problema de la mochila representa uno de los desafíos más conocidos dentro del ámbito de la optimización combinatoria, y aparece frecuentemente en aplicaciones reales como la asignación de recursos, la administración de inventarios y la toma de decisiones en inversiones. Este problema consiste en seleccionar un conjunto de objetos que deben colocarse dentro de una mochila con capacidad limitada, de modo que el valor total de los objetos seleccionados sea el mayor posible sin superar el límite de volumen o peso permitido. Para resolver este problema existen diversas estrategias, que incluyen tanto métodos exactos como aproximaciones basadas en heurísticas. En este trabajo se abordarán dos enfoques particulares, uno de ellos es la búsqueda exhaustiva y por otro lado los algoritmos heurísticos.

La búsqueda exhaustiva implica analizar todas las combinaciones posibles de elementos con el fin de identificar la mejor solución al problema. Esta técnica garantiza eoncontrar la solución óptima, ya que considera todas las alternativas viables. En el contexto del problema de la mochila, esto significa evaluar cada subconjunto de objetos para determinar cuál maximiza el valor total sin exceder la capacidad de peso. Sin embargo, su principal desventaja es el elevado costo computacional, ya que la cantidad de combinaciones posibles crece exponencialmente con el número de objetos disponibles. Por este motivo, su aplicación resulta poco práctica para instancias de gran tamaño, haciendo inviable su uso en problemas con muchos elementos debido al tiempo y recursos necesarios.

Los métodos heurísticos, por otro lado, ofrecen soluciones aproximadas mediante procesos más eficientes. Aunque no aseguran encontrar la solución óptima, permiten obtener resultados aceptables en tiempos considerablemente menores, lo que los hace prácticos para problemas de gran escala. Estas técnicas se basan en reglas empíricas o criterios estratégicos para guiar la selección de soluciones. En particular, en este informe se utilizará el algoritmo greedy, una heurística que construye la solución en forma progresiva, eligiendo en cada paso el elemento que proporciona el mayor beneficio inmediato según el criterio de optimización utilizado. Si bien no siempre alcanza la solución ideal, este tipo de algoritmo es muy útil para tratar problemas complejos y de gran escala gracias a su rapidez y simplicidad.

1.1. Enunciado del Trabajo

Problema de la mochila Consiste en elegir, de entre un conjunto de n elementos, (cada uno con un valor $i, y un volumen Vi), aquellos que puedan ser cargados en una mochila de volumen V de manera que el valor obtenido sea máximo.

Utilizando una computadora, resolver el siguiente problema:

Cuáles son los elementos de la lista siguiente que cargaremos en una mochila de 4200 cm3 de manera que su valor en $ sea máximo.

Objeto	Volumen (cm^3.)	Valor ($)
1	150	20
2	325	40
3	600	50
4	805	36
5	430	25
6	1200	64
7	770	54
8	60	18
9	930	46
10	353	28

Volumen máximo soportado por la mochila: 4200 cm^3.

Para su solución, utilizar un procedimiento exhaustivo que consiste en evaluar para cada subconjunto de elementos el valor correspondiente y, posteriormente, clasificando los subconjuntos por su valor de mayor a menor, encontrar cuál es el subconjunto solución.

Ejercicios:

1. Resolver el problema de la Mochila utilizando una búsqueda exhaustiva.
2. Resolver el ejercicio anterior usando el algoritmo greedy y comentar su similitud o no con el exhaustivo.

3. Plantear el problema de la mochila teniendo en cuenta los pesos en lugar del volumen. Luego, dados 3 elementos, cuyos pesos son:
   - 1800 grs.,
   - 600 grs.,
   - y 1200 grs.

   y cuyos valores son:
   - $72,
   - $36
   - y $60 respectivamente, y con una mochila que puede soportar hasta **3000 grs**.

A. Hallar una solución **utilizando un algoritmo goloso y exhaustivo**

B. **Analizar dicha solución respecto a su grado de optimización** y elaborar las **conclusiones** que considere adecuadas.

1.2. Conceptos teóricos

Óptimo Global: La mejor solución posible dentro de todo el espacio de búsqueda. En el contexto del problema de la mochila, corresponde a la combinación de elementos que brinda el valor total más alto sin exceder la capacidad.

Óptimo Local: Una solución que es mejor que sus vecinas inmediatas, pero no necesariamente la mejor del conjunto total. Algoritmos como el greedy pueden caer en óptimos locales si toman decisiones que parecen buenas en el momento pero que impiden alcanzar una mejor solución final.

Espacio de Soluciones: Conjunto de todas las combinaciones posibles de elementos que pueden incluirse (o no) en la mochila. Cada combinación representa un punto distinto en este espacio.

Heurística: Método de resolución que busca encontrar soluciones satisfactorias de forma eficiente, sacrificando en muchos casos la garantía de obtener la solución óptima. Son útiles cuando el espacio de soluciones es muy grande o cuando se requiere una respuesta en tiempos reducidos.

Combinatoria: Rama de las matemáticas que estudia las posibles formas de seleccionar y organizar elementos dentro de un conjunto. En el problema de la mochila, la combinatoria determina la cantidad total de soluciones posibles (es decir, el tamaño del espacio de soluciones), que se incrementa exponencialmente a medida que aumenta la cantidad de elementos disponibles.

2. Descripción de la metodologías de desarrollo abordadas

El problema de la mochila (o Knapsack Problem) es una problemática clásica de optimización combinatoria que busca seleccionar un subconjunto de elementos con el objetivo de maximizar un beneficio total (por ejemplo, valor económico), respetando una o más restricciones (como peso o volumen máximo permitido). Este tipo de problema aparece en numerosos contextos reales, desde la logística hasta la asignación de recursos. Para su resolución, existen múltiples enfoques que varían en su eficiencia y capacidad de encontrar soluciones óptimas.

Este trabajo se centra en la implementación, análisis y comparación de dos metodologías bien diferenciadas para abordar esta problemática: la Búsqueda Exhaustiva, que evalúa todas las combinaciones posibles para garantizar una solución óptima, y el Algoritmo Greedy, que se basa en decisiones locales rápidas para construir soluciones de forma eficiente. Ambas estrategias permiten observar cómo varía el rendimiento, el tiempo de ejecución y la calidad de las soluciones según el enfoque utilizado.

2.1. Búsqueda Exhaustiva

La búsqueda exhaustiva es una técnica que evalúa todas las combinaciones posibles de elementos para encontrar la solución óptima al problema. En el contexto de la mochila, esto implica analizar todos los subconjuntos de elementos y calcular para cada uno el valor total y las restricciones impuestas (como el peso o el volumen), conservando aquel que maximice el beneficio sin violar las limitaciones.

Aunque este método garantiza la obtención de la mejor solución posible, su principal desventaja es el tiempo de ejecución, que crece exponencialmente con la cantidad de elementos disponibles. Esto lo vuelve ineficiente e inviable para instancias de gran tamaño, pero es útil como referencia para comparar otros algoritmos heurísticos o aproximados.

2.2. Algoritmo Greedy

El algoritmo Greedy, o voraz, es una estrategia heurística que construye una solución de forma secuencial, eligiendo en cada paso el elemento que aporta el mayor beneficio inmediato, en general calculado a través de la relación valor/peso o valor/volumen. De esta manera, se toma una decisión local óptima en cada iteración con la esperanza de que dicha elección conduzca a una solución cercana a la óptima global.

Su principal ventaja es la eficiencia: es simple de implementar y extremadamente rápido en comparación con métodos exhaustivos. Sin embargo, su limitación está en que no siempre garantiza una solución óptima, ya que no revisa combinaciones globales ni corrige elecciones previas. Aun así, es especialmente útil cuando se necesita una solución rápida y aceptablemente buena en escenarios con restricciones de tiempo.

3. Descripción de las herramientas de programación utilizadas

3.1. Lenguaje de Programación

El programa está desarrollado utilizando Python, un lenguaje de programación de alto nivel, interpretado y de propósito general. La elección de Python se debe a su sintaxis clara y legible, lo que agiliza el desarrollo y mantenimiento del código, especialmente en la implementación de algoritmos complejos como los genéticos. Además, su amplia variedad de librerías lo convierte en una opción robusta para tareas que involucran cálculos numéricos, manipulación de datos y visualización.

3.2. Librerías utilizadas

Para el desarrollo de este trabajo práctico se ha empleado la siguiente librería de Python:

- **time**: Esta es una librería estándar de Python utilizada para medir tiempos de ejecución. En este trabajo, se emplea para analizar la eficiencia de cada algoritmo implementado (búsqueda exhaustiva y algoritmo greedy), permitiendo comparar sus tiempos de resolución.
  - $time.time()$: si bien no fue utilizada en este trabajo, es una función comúnmente empleada para medir tiempos de ejecución. Retorna el tiempo actual en segundos con precisión suficiente para procesos relativamente largos. Sin embargo, no es adecuada para medir tiempos muy breves, ya que su resolución puede no ser suficiente para detectar diferencias menores a 1 milisegundo.
  - $time.perf\_counter()$: devuelve el tiempo en segundos con alta resolución (incluyendo fracciones de microsegundo), ideal para medir tareas muy rápidas incluso cuando la diferencia temporal es mínima. Su uso consiste en guardar el instante de inicio y fin de un bloque de código, y restarlos para obtener el tiempo total de ejecución.

4. Descripción de la forma de trabajo abordada en equipo

Dado el contexto de virtualidad en el que se desarrolló el trabajo práctico, se optó por una organización colaborativa y estructurada del equipo mediante diferentes herramientas digitales. La comunicación diaria se sostuvo mayormente vía WhatsApp para una coordinación ágil, y realizamos llamadas mediante Discord para compartir archivos y trabajar en el código. Asimismo, abordamos el informe mediante la plataforma Overleaf en formato LaTeX.

El código se desarrolló utilizando el editor de código Visual Studio Code, lo que facilitó la colaboración simultánea en el código fuente, el cual fue implementado en lenguaje Python, como fue mencionado anteriormente. Principalmente la forma de trabajo fue conjunta, lo que si bien inicialmente pudo resultar algo lento, rápidamente luego nos permitió avanzar en el desarrollo del código, ya que al estar trabajando todos al mismo tiempo en comunicación se pudieron resolver dudas y errores con mayor facilidad, y aprender más rápido a partir de ellos.

5. Código del programa

```
import time
VOL_MOCHILA = 4200
PESO_MOCHILA = 3000
LISTA1 = [[150, 20], [325, 40], [600, 50], [805, 36], [430,25], [1200, 64], [770, 54], [60, 18],
    [930, 46],[353, 28]] # Volumen y Valor
LISTA2 = [[1800, 72], [600, 36], [1200, 60]] # Peso y Valor
```

Importamos la librería 'time' para ver los tiempos de ejecución de cada método. Además, hacemos la definición de constantes del programa. La constante 'VOL_MOCHILA' almacena la restricción del volumen del problema original, mientras que la constante 'PESO_MOCHILA' almacena la restricción de peso en la redefinición del problema (apartado 3). Por su parte, se definen las dos listas de objetos, con sus respectivos volumenes o pesos según corresponda. Utilizar estas variables globales permite reutilizar este programa haciendo cambios mínimos.

```
def menu_ppal():
    global ELEMENTOS
    global SELECCION
    while True:
        print("\n--- Cuantos elementos tiene tu mochila? ---")
        print("1. 10 elementos (usando volumen)")
        print("2. 3 elementos (usando peso)")

        opcion = input("Elegí una opción (1 o 2): ")

        if opcion == "1":
            ELEMENTOS = 10
            break
        elif opcion == "2":
            ELEMENTOS = 3
            break
        else:
            print("Opción inválida. Probá de nuevo.\n")

    while True:
        print("\n--- Tipo de algoritmo disponibles ---")
        print("1. Búsqueda exhaustiva")
        print("2. Método Greedy")

        opcion = input("Elegí una opción (1 o 2): ")

        if opcion == "1":
            SELECCION = 'E'
            break
        elif opcion == "2":
            SELECCION = 'G'
            break
        else:
            print("Opción inválida. Probá de nuevo.\n")
```

Definimos la función 'menu_ppal', que se llama al comienzo de la ejecución del programa para especificar la cantidad de objetos que tendrá el listado (10 utilizando volumen o 3 utilizando peso) y el tipo de algoritmo deseado (búsqueda exhaustiva o algoritmo greedy). Se definen las variables globales 'ELEMENTOS', que podrá asumir 10 o 3 como valor y que se utilizará más adelante en nuestro programa principal; y 'SELECCION', que podrá asumir E o G como valor y que se utilizará al momento de abordar un algoritmo en el programa principal. Luego, se utilizan dos estructuras de bucle que se repiten hasta que se ingrese una opción válida de las mencionadas anteriormente.

```
def busqueda_exhaustiva(lista, capacidad, tamano):
    mejor_valor = 0
    mejor_comb = []

    for i in range(2**tamano):
        combinacion = []
        volumen_total = 0
        valor_total = 0

        for j in range(tamano):
            if (i >> j) & 1 == 1:
                volumen_total += lista[j][0]
                valor_total += lista[j][1]
                combinacion.append(j + 1)
```

```
            """
            La condición permite desplazar i hacia la derecha j bits. Por ejemplo:

            i = 1       # 001 en binario
            combinacion = []

            j = 0       --> i >> 0 = 001
                combinacion = [[1800, 72], ]
                volumen_total = 1800
                valor_total = 72
            j = 1       --> i >> 1 = 00
                combinacion = [[1800, 72], ]
            j = 2       --> i >> 2 = 0
                combinacion = [[1800, 72], ]

            combinacion = [[1800, 72], ]
            volumen_total = 1800
            valor_total = 72

            mejor_valor = valor_total
            mejor_comb = combinacion[:]
            """

        # Actualizar mejor combinación válida
        if volumen_total <= capacidad and valor_total > mejor_valor:
            mejor_valor = valor_total
            mejor_comb = combinacion[:]

    return mejor_valor, mejor_comb
```

Definimos la función busqueda_exhaustiva, que recibe como parámetro una lista (arreglo bidimensional que contiene el valor y el volumen/peso de cada objeto), la capcidad (un número en base a si debe restringir por volumen o por peso), y el tamaño de la lista recibida; y que devuelve un arreglo que posee la mejor combinación (óptima global) y el valor asociado a esa mejor combinación obtenida.

Primeramente, inicializamos en cero en valor de la mejor combinación y en vacío el arreglo de dicha mejor combinación. A partir de eso, se itera tantas veces como cantidad de combinaciones posibles (en base a la lista en uso). Allí se declara un arreglo combinación que será usado para armar y comparar cada posibilidad, y luego el volumen y el valor total (usado según corresponda).

Luego, se itera nuevamente tantas veces como elementos posea la lista en uso. En cada iteración, se toma el número i (número de combinación actual, en decimal) y haciendo uso del operador '$\gg$' se lo transforma a binario y se lo desplaza j veces a la derecha, para luego quedarse sólo con el bit menos significativo (el que está más a la derecha) utilizando el operador '&'. Entonces, la condición evalúa si ese bit es 1. En caso de ser así, dicho objeto debería incluirse en la combinación actual: se suma el volumen/peso y el valor de dicho item en el acumulador y dicho item j se añade en la combinación actual.

Sabiendo dichas variables almacenadas para la combinación en curso, se verifica si el volumen/peso y el valor total son más óptimos que los que se tenían hasta el momento, de ser así son almacenados como nuevos mejores valores. Esto se repite para cada combinación posible de todo el espacio solución.

Este proceso iterativo permite realizar todas las combinaciones posibles a partir de la representación binaria de los números para saber cuál objeto incluir y cual no en cada subconjunto (combinación).

```
def algoritmo_greedy(lista, capacidad):
    # Agregar índice y ratio valor/volumen (o bien valor/peso)
    objetos_con_ratio = []
    for idx in range(len(lista)):
        volumen = lista[idx][0] # en el caso de 3 elementos, decimos peso en vez de volumen pero
    sirve igual
        valor = lista[idx][1]
```

```
        ratio = valor / volumen
        objetos_con_ratio.append((idx, volumen, valor, ratio))

    # Ordenar por mayor valor relativo (valor/volumen)
    objetos_ordenados = sorted(objetos_con_ratio, key=lambda x: x[3], reverse=True)

    mochila = []
    capacidad_actual = 0
    valor_total = 0

    for i, volumen, valor, ratio in objetos_ordenados:
        if capacidad_actual + volumen <= capacidad:
            mochila.append(i)
            capacidad_actual += volumen
            valor_total += valor
    return mochila, valor_total, capacidad_actual
```

Definimos la función algoritmo_greedy, que recibe como parámetro una lista (arreglo bidimensional que contiene el valor y el volumen/peso de cada objeto) y la capacidad (un número en base a si debe restringir por volumen o por peso); y que devuelve un arreglo que posee la mochila obtenida, el valor asociado a esa mochila y la capacidad que ocupan los objetos de la mochila.

Primeramente, se inicializa un arreglo vacío donde se guardarán cada uno de los objetos de la lista, incluyendo su ratio (valor / volumen, o valor / peso). A continuación se itera recorriendo la lista para completar dicho arreglo con los valores y ratio de cada objeto.

Luego se ordena el arreglo anterior por el criterio del ratio, de mayor a menor. Se declaran también variables acumulador para la capacidad actual y el valor total, y se declara en vacío el arreglo de la mochila a devolver. Finalmente se itera en orden según el arreglo ordenado. Para cada iteración, en caso que aún no se haya superado la capacidad máxima de la mochila (ya sea en volumen o peso, según se trabaje), se agrega el objeto actual a la mochila y se añade sus respectivos valores a los acumuladores.

De esta manera, se obtiene una mochila que no supera la capacidad máxima y que prioriza los objetos que tengan una mejor razón respecto al valor que poseen por su volumen/peso, o sea que se priorizan aquellos objetos que aportan el mayor beneficio inmediato (óptimo local).

```python
def main():
    menu_ppal()
    if (ELEMENTOS == 10):
        n = len(LISTA1)
        lista = LISTA1
    else:
        n = len(LISTA2)
        lista = LISTA2

    if (SELECCION == 'E'):
        if (ELEMENTOS == 10):
            start_time = time.perf_counter()
            mejor_valor, mejor_comb = busqueda_exhaustiva(lista, VOL_MOCHILA, n)
            elapsed_time = time.perf_counter() - start_time

            print (f"Tiempo de ejecución: {elapsed_time:.6f} segundos | Ejecutadas {2**n} combinaciones en la Busqueda Exhaustiva" )
            print (f"La mejor combinación válida encontrada posee {len(mejor_comb)} objetos, existiendo {n} en el listado completo original")

            print("Elementos de la mejor combinación válida encontrada:")
            for idx in mejor_comb:
                print(f"Objeto N{idx + 1} - Volumen: {lista[idx][0]}, Valor: ${lista[idx][1]}")
            print(f"Valor total: ${mejor_valor}")
            print(f"Volumen total: {sum([lista[i][0] for i in mejor_comb])} cm")

        else:
            start_time = time.perf_counter()
            mejor_valor, mejor_comb = busqueda_exhaustiva(lista, PESO_MOCHILA, n)
            elapsed_time = time.perf_counter() - start_time
            print (f"Tiempo de ejecución: {elapsed_time:.6f} segundos | Ejecutadas {2**n} combinaciones en la Busqueda Exhaustiva" )
            print (f"La mejor combinación válida encontrada posee {len(mejor_comb)} objetos, existiendo {n} en el listado completo original")

            print("Elementos de la mejor combinación válida encontrada:")
            for idx in mejor_comb:
                print(f"Objeto N{idx + 1} - Peso: {lista[idx][0]}g, Valor: ${lista[idx][1]}")
            print(f"Valor total: ${mejor_valor}")
            print(f"Peso total: {sum([lista[i][0] for i in mejor_comb])} gramos")

    else:
        if (ELEMENTOS == 10):
            start_time = time.perf_counter()
            mochila, valor_total, volumen_total = algoritmo_greedy(lista, VOL_MOCHILA)
            elapsed_time = time.perf_counter() - start_time

            print(f"Tiempo de ejecución: {elapsed_time:.6f} segundos | Algoritmo Greedy (valor/volumen)")

            for i in mochila:
                print(f"Objeto N{i + 1} - Volumen: {lista[i][0]} cm, Valor: ${lista[i][1]}")
            print(f"Valor total: ${valor_total}")
            print(f"Volumen total: {volumen_total} cm")

        else:
            start_time = time.perf_counter()
            mochila, valor_total, peso_total = algoritmo_greedy(lista, PESO_MOCHILA)
            elapsed_time = time.perf_counter() - start_time

            print(f"Tiempo de ejecución: {elapsed_time:.6f} segundos | Algoritmo Greedy (valor/peso)")

            for i in mochila:
```

```
                print(f"Objeto N{i + 1} - Peso: {lista[i][0]} g, Valor: ${lista[i][1]}")
            print(f"Valor total: ${valor_total}")
            print(f"Peso total: {peso_total} g")


if __name__ == "__main__":
    main()
```

Por último, en la función main() se llama al método correspondiente al menú principal en el cual se definian los criterios de evaluación. En base a esas elecciones se bifurca el código si es una lista de 10 o 3 elementos y el tipo de búsqueda.

Finalmente para la obtención de cada conjunto de valores finales se muestra cada objeto con su volumen o peso y el valor correspondiente, se detalla el valor total seguido de el peso total.

6. Salidas por pantalla de las corridas

6.1. Búsqueda Exhaustiva con 10 elementos (usando volumen)

```
Tiempo de ejecución: 0.001196 segundos | Ejecutadas 1024 combinaciones en la Busqueda Exhaustiva
La mejor combinación válida encontrada posee 8 objetos, existiendo 10 en el listado completo
    original
Elementos de la mejor combinación válida encontrada:
Objeto N1 - Volumen: 150, Valor: $20
Objeto N2 - Volumen: 325, Valor: $40
Objeto N3 - Volumen: 600, Valor: $50
Objeto N5 - Volumen: 430, Valor: $25
Objeto N6 - Volumen: 1200, Valor: $64
Objeto N7 - Volumen: 770, Valor: $54
Objeto N8 - Volumen: 60, Valor: $18
Objeto N10 - Volumen: 353, Valor: $28
Valor total: $299
Volumen total: 3888 cm
```

6.2. Algoritmo Greedy con 10 elementos (usando volumen)

```
Tiempo de ejecución: 0.000034 segundos | Algoritmo Greedy (valor/volumen)
Objeto N8 - Volumen: 60 cm, Valor: $18
Objeto N1 - Volumen: 150 cm, Valor: $20
Objeto N2 - Volumen: 325 cm, Valor: $40
Objeto N3 - Volumen: 600 cm, Valor: $50
Objeto N10 - Volumen: 353 cm, Valor: $28
Objeto N7 - Volumen: 770 cm, Valor: $54
Objeto N5 - Volumen: 430 cm, Valor: $25
Objeto N6 - Volumen: 1200 cm, Valor: $64
Valor total: $299
Volumen total: 3888 cm
```

6.3. Búsqueda Exhaustiva con 3 elementos (usando peso)

```
Tiempo de ejecución: 0.000027 segundos | Ejecutadas 8 combinaciones en la Busqueda Exhaustiva
La mejor combinación válida encontrada posee 2 objetos, existiendo 3 en el listado completo
    original
Elementos de la mejor combinación válida encontrada:
Objeto N1 - Peso: 1800g, Valor: $72
Objeto N3 - Peso: 1200g, Valor: $60
Valor total: $132
Peso total: 3000 gramos
```

6.4. Algoritmo Greedy con 3 elementos (usando peso)

```
Tiempo de ejecución: 0.000028 segundos | Algoritmo Greedy (valor/peso)
Objeto N2 - Peso: 600 g, Valor: $36
Objeto N3 - Peso: 1200 g, Valor: $60
Valor total: $96
Peso total: 1800 g
```

7. Conclusión

Al analizar los resultados, se observó que, con una base de 10 elementos, tanto el método exhaustivo como el heurístico llegaron a la misma solución. Sin embargo, al reducir la base inicial a 3 elementos, las soluciones obtenidas por ambos métodos fueron distintas.

La búsqueda exhaustiva tiene la ventaja de garantizar siempre la solución óptima, ya que evalúa todas las combinaciones posibles. No obstante, su principal desventaja es la ineficiencia computacional, pues el tiempo de ejecución crece de forma exponencial con el tamaño del problema, volviéndose impracticable para instancias grandes.

Por otro lado, los métodos heurísticos, como el algoritmo Greedy, ofrecen una alternativa más eficiente para problemas donde la búsqueda exhaustiva no es factible. Aunque no aseguran obtener la mejor solución, suelen entregar resultados de buena calidad en tiempos mucho menores, lo que los hace apropiados para problemas de gran escala o cuando los recursos computacionales son limitados.

En términos de rendimiento, al utilizar 10 objetos, el algoritmo exhaustivo fue un 3700 % más lento (38 veces más) que el algoritmo Greedy. En cambio, con 3 objetos, la diferencia fue mínima: apenas un 4 % más lento (1,04 veces). Esto evidencia que la brecha de eficiencia entre ambos métodos crece drásticamente a medida que aumenta el tamaño del problema.

En conclusión, la elección del método depende del tamaño y la complejidad del problema, así como de las restricciones de tiempo y recursos. La búsqueda exhaustiva es recomendable para problemas pequeños donde se requiere la máxima precisión, mientras que las heurísticas resultan más adecuadas en escenarios donde la rapidez y la eficiencia son prioritarias, aunque se acepte una solución aproximada.

8. Bibliografía

[1] GeeksforGeeks. “Problema de la mochila”. https://www.geeksforgeeks.org/dsa/0-1-knapsack-problem-dp-10/.

[2] GeeksforGeeks. “Función time.perf_counter() en Python”. https://www.geeksforgeeks.org/python/time-perf_counter-function-in-python/.

[3] Python Documentation. “time — Acceso y conversiones de tiempo”. https://docs.python.org/3/library/time.html.

[4] W3Schools. “Problema de la mochila”. https://www.w3schools.com/dsa/dsa_ref_knapsack.php.

[5] Wikipedia. “Problema de la mochila”. https://en.wikipedia.org/wiki/Knapsack_problem.