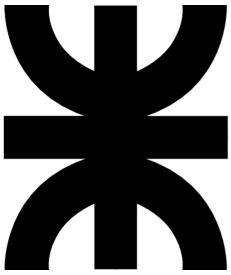


UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL ROSARIO

Materia: Algoritmos Genéticos **Comisión:** 3EK03

PROBLEMA DE LA MOCHILA

Trabajo Práctico N°2

Ciclo Lectivo: 2025

Integrantes del grupo:

Nombre y Apellido	Correo electrónico	Legajo
Juan Cruz Mondino	juancm.2000@hotmail.com	51922
Alexis Mateo	alexisjoelmateo@gmail.com	51191
Gustavo Giampietro	gustgiam2001@gmail.com	50671

Profesores:

Daniela Díaz
Víctor Lombardo

Índice

1. Enunciado del Problema de la Mochila

El problema consiste en elegir, de entre un conjunto de n elementos (cada uno con un valor $\$_i$ y un volumen V_i), 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 $4200\,\text{cm}^3$ 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

Tabla 1: Elementos disponibles para la mochila

Volumen máximo soportado por la mochila: $4200\ \text{cm}^3$.

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.

1.1. Ejercicio 1

Resolver el problema de la Mochila utilizando una búsqueda exhaustiva.

1.2. Ejercicio 2

Resolver el ejercicio anterior usando el algoritmo greedy y comentar su similitud o no con el exhaustivo.

1.3. Ejercicio 3

Dados 3 elementos, plantear el problema de la mochila teniendo en cuenta los pesos en lugar del volumen. La pregunta correspondiente a este planteo es la siguiente:

¿Cuáles son los elementos de la lista siguiente que cargaremos en una mochila de 3000 gramos de manera que su valor en \$ sea máximo?

Objeto	Volumen (cm^3)	Valor ($)
1	1800	72
2	600	36
3	1200	60

Tabla 2: Elementos disponibles para la mochila

Peso máximo soportado por la mochila: 3000 gramos.

1.3.1. Apartado A

Hallar una solución utilizando un algoritmo goloso y exhaustivo.

1.3.2. Apartado B

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

2. Metodología de Desarrollo

El programa se ejecuta con la siguiente sintaxis:

```
python TP2_AG_G2.py -e p -f v
```

Código 1: Comando utilizado para el abordaje de los ejercicios 1 y 2

Donde:

- `-e`: **Elección de elementos**:
  - `p`: Conjunto predefinido de elementos
  - `a`: Generación aleatoria de elementos
  - `c`: Carga manual por consola
- `-f`: **Factor de decisión**:
  - `v`: Optimización por volumen
  - `p`: Optimización por peso

En ese caso, se realizarían los ejercicios 1 y 2, debido a se que realiza una búsqueda exhaustiva y una búsqueda heurística de la elección de los elementos que cabrían en la mochila, teniendo esta última una capacidad máxima (siendo 4200 cm^3 el valor propuesto).

En el ejercicio 3 se realiza una búsqueda exhaustiva y una búsqueda heurística para realizar la selección de los elementos en función del peso de cada uno de ellos, donde en este caso la mochila tiene un peso máximo que no puede superar (siendo 3000 gramos el valor propuesto por el ejercicio).

Para poder desarrollar lo recientemente mencionado se ejecuta la siguiente sintaxis:

```
python TP2_AG_G2.py -e p -f p
```

Código 2: Comando utilizado para el abordaje del ejercicio 3

Esta información, ingresada por la línea de comando, servirá para tomar la decisión de elegir el factor de decisión por el cual se realizará la búsqueda de la solución óptima. Además, se ingresa el modo de generar los elementos disponibles para realizar la optimización, los cuales, parte de ellos, formarán la solución óptima del problema de la mochila.

El factor de decisión, ya sea el peso o el volumen, no es un dato de gran importancia para la resolución del problema como tal, sino que sirve para que la experiencia del usuario al utilizar el programa sea lo más comprensible posible. Es decir, el factor de decisión simplemente sirve para dar a entender que la variable del problema de optimización es el peso o la capacidad volumétrica que soporta la mochila.

Por otro lado, se ingresa la manera en la que se generan los datos de los elementos en los que se basará el problema. Existen tres formas distintas de generar los datos de cada uno de los elementos. En primer lugar, los datos preestablecidos, que son los propuestos por el enunciado del problema. En segundo lugar, los datos generados a mano, los que servirían para llevar a la práctica el programa sin tener que modificar el código. Y por último, los datos se podrían generar de forma aleatoria para poder estudiar rápidamente los distintos resultados al correr el programa varias veces seguidas y de forma continua. De igual manera, es posible generar el peso o volumen máximo de la mochila.

Una vez generados y presentados los elementos disponibles, el programa aplica dos métodos para encontrar la solución al problema: una búsqueda exhaustiva y una búsqueda heurística (algoritmo Greedy). En el primer caso, se evalúan todas las combinaciones posibles de elementos, verificando para cada una si cumple con la restricción de capacidad. De las combinaciones factibles, se identifica aquella que maximiza el valor total (precio) como solución óptima. El resultado incluye el listado de elementos seleccionados, su medida total y su valor, así como el tiempo de ejecución de este método. Además, se genera un gráfico que muestra la distribución de combinaciones según la cantidad de elementos y destaca visualmente la solución óptima.

En el segundo caso, el algoritmo heurístico ordena los elementos según su relación valor/medida, seleccionando de forma secuencial aquellos que mejoran el valor de la mochila sin exceder la capacidad máxima. Aunque este método no garantiza encontrar la solución global óptima, permite obtener rápidamente una aproximación de buena calidad, lo que lo hace útil para comparar velocidad y rendimiento frente a la búsqueda exhaustiva. Este método se aproxima mucho más a lo que realiza una persona cuando se encuentra en esta situación debido a que ésta selecciona los elementos con un criterio similar, pero en función de su experiencia y su instinto, sin estudiarlo a fondo.

Finalmente, el programa presenta al usuario los resultados de ambos métodos, mostrando las diferencias en el valor obtenido, la composición de la mochila y los tiempos de ejecución, lo que permite analizar el impacto del enfoque utilizado en la calidad de la solución y en el costo computacional.

3. Forma de Trabajo Abordada por el Grupo

Para desarrollar la simulación, nuestro equipo optó por implementar el algoritmo en **Python**, utilizando como entorno de desarrollo principal **Visual Studio Code (VS Code)**.

Dado que trabajamos de manera distribuida, cada integrante programó desde su propio equipo. Para facilitar la colaboración, empleamos las siguientes herramientas:

- **Google Drive**: Donde organizamos carpetas y documentos compartidos, permitiendo ediciones simultáneas y acceso centralizado a los recursos del proyecto.
- **GitHub**: Utilizamos este sistema de control de versiones para coordinar el desarrollo del código sin necesidad de intercambiar archivos manualmente. Además, aprovechamos sus funcionalidades para mantener un historial de cambios y garantizar la seguridad del trabajo.

Esta combinación de herramientas nos permitió trabajar de manera eficiente y coordinada, asegurando la consistencia del proyecto en todo momento.

4. Herramientas de Programación Utilizadas

El desarrollo de este proyecto requirió el uso de varias librerías fundamentales de Python, cada una con un propósito específico:

Manejo del sistema

- `os`: Utilizada para operaciones del sistema operativo, principalmente para limpiar la pantalla de la terminal, luego de que se haya detectado si el sistema es Windows o Unix/Linux.
- `sys`: Fundamental para manejar los argumentos de línea de comandos (`sys.argv`), permitiendo configurar el comportamiento del programa sin modificar el código fuente o tener que ingresar la decisión por pantalla. También se empleó para terminar la ejecución con `sys.exit()` en caso de errores.

Generación y manipulación de datos

- `random`: Proporciona generación de números aleatorios para crear elementos con volúmenes (o pesos) y precios aleatorios cuando se selecciona el modo correspondiente (`random.randint()`).
- `itertools.combinations`: Función crítica para la búsqueda exhaustiva, la cual genera todas las combinaciones posibles de elementos que podrían conformar soluciones al problema de la mochila.

Medición de tiempo y visualización

- `time.perf_counter`: Permite medir con alta precisión el tiempo de ejecución de los algoritmos utilizando la función `perf_counter()`, siendo fundamental para comparar el rendimiento entre los diferentes enfoques de búsqueda de la solución óptima.
- `matplotlib.pyplot`: Librería clave para la visualización de resultados, utilizada para generar gráficos de barras que muestran la distribución de soluciones factibles y no factibles de uno de los tipos de búsqueda, incluyendo anotaciones personalizadas.

5. Código

El programa cuenta con diversas funciones que encapsulan operaciones específicas, facilitando su lectura, mantenimiento y reutilización. En esta sección se explicará cada una de las funciones del código Python elaborado.

5.1. Importación de librerías utilizadas

Esta sección contiene todas las librerías importadas para la elaboración del código. Cada una de estas fue explicada y desarrollada en profundidad en la sección anterior. El código de las librerías importadas es el siguiente:

```
1 import os
2 import sys
3 import random
4 import matplotlib.pyplot as plt
5 from itertools import combinations
6 from time import perf_counter
```

Código 3: Importación de librerías

5.2. Limpieza de Pantalla

Esta funcionalidad tiene como finalidad borrar el contenido de la consola para mejorar la claridad y organización durante la interacción con el usuario. Su implementación utiliza el siguiente enfoque multiplataforma:

```
1 def limpiar_pantalla():
2     if os.name == 'nt':
3         os.system('cls')
4     else:
5         os.system('clear')
```

Código 4: Implementación de la función que limpiar la consola

- **Detección del sistema operativo**: Mediante `os.name`, que retorna `'nt'` para Windows y `'posix'` para sistemas Unix-like (Linux, MacOS).
- **Ejecución del comando apropiado**:
  - `cls` en sistemas Windows
  - `clear` en sistemas Unix/Linux

Esta implementación mejora la experiencia del usuario al mantener la terminal libre de contenido previo antes de mostrar nueva información.

5.3. Ingreso y Validación de Argumentos

La función `ingreso_argumentos()` encapsula la validación y lectura de los parámetros que el usuario pasa al ejecutar el programa desde la línea de comandos. Esta se muestra a continuación:

```python
def ingreso_argumentos():
    if len(sys.argv) != 5 or sys.argv[1] != "-e" or sys.argv[3]
        != "-f":
        print("Uso: python TP2_AG_G2.py -e <eleccion de
           elementos: p-predefinido a-aleatorio, c-cargar por
           consola> -f <factor de decision: v-volumen p-peso>")
        sys.exit(1)
    if (sys.argv[2] != "p" and sys.argv[2] != "a" and sys.argv
       [2] != "c") or (sys.argv[4] != "v" and sys.argv[4] != "p
       "):
        print("Error: python TP2_AG_G2.py -e <eleccion de
           elementos: p-predefinido a-aleatorio, c-cargar por
           consola> -f <factor de decision: v-volumen p-peso>")
        sys.exit(1)
    return sys.argv[2], sys.argv[4]
```

Código 5: Implementación del ingreso y la validación de argumentos por línea de comandos

El formato correcto sería `-e` seguido de la elección de elementos (`p`, `a` o `c`) y `-f` seguido del factor de decisión (`v` o `p`). Si la estructura o los valores son inválidos, muestra el mensaje de uso y finaliza el programa. Si todo es correcto, devuelve la elección de elementos y el factor de decisión para que el flujo principal configure el problema.

5.4. Ingreso de un entero

Esta funcionalidad se encarga de solicitar y validar la entrada de un número entero dentro de un rango específico.

```python
def pedir_entero(mensaje, minimo, maximo):
    while True:
        try:
            valor = int(input(mensaje))
            if minimo <= valor <= maximo:
                return valor
            else:
                print(f"Error: Ingrese un valor entre {minimo}
                   y {maximo}.")
                input("Presione enter para continuar...")
                print()
        except ValueError:
            print("Error: Ingrese un numero entero.")
            input("Presione enter para continuar...")
            print()
```

Código 6: Implementación de una función para solicitar un entero

Emplea un ciclo `while` que solo finaliza cuando el dato ingresado es válido. Dentro del bloque `try`, intenta convertir el valor a entero, y si la conversión falla, captura la excepción `ValueError` y

muestra un mensaje de error. En caso de que el número no cumpla con las restricciones de rango, también emite un mensaje y solicita nuevamente el ingreso. Este enfoque garantiza que el flujo del programa no se vea interrumpido por errores de tipeo o entradas inválidas.

5.5. Generación de Elementos

Este procedimiento construye la lista de elementos que participarán en el problema. Cada elemento se representa como una lista con tres datos:

- Número identificador
- Medida (peso o volumen)
- Precio

```
def generar_elementos(cantidad_elementos, capacidad_maxima,
    precio_max_elem, es_aleatorio, factor_decision):
    elementos = [[] for _ in range(cantidad_elementos)]
    for i in range(cantidad_elementos):
        if es_aleatorio:
            medida = random.randint(1, capacidad_maxima)
            precio = random.randint(1, precio_max_elem)
        else:
            if factor_decision == "v":
                print(f"Ingrese el volumen del elemento {i + 1}
                    (maximo {capacidad_maxima}): ", end="")
            else:
                print(f"Ingrese el peso del elemento {i + 1} (
                    maximo {capacidad_maxima}): ", end="")
            medida = pedir_entero("", 1, capacidad_maxima)
            print(f"Ingrese el precio del elemento {i + 1} (
                maximo {precio_max_elem}): ", end="")
            precio = pedir_entero("", 1, precio_max_elem)
            print()
        elementos[i] = [i + 1, medida, precio]
    return elementos
```

Código 7: Implementación de la función que genera los elementos disponibles

El procedimiento varía según el parámetro `es_aleatorio`:

- Si es `True`, se generan medidas y precios aleatorios dentro de los límites establecidos
- Si es `False`, el usuario ingresa cada valor manualmente mediante llamadas a `pedir_entero()`.

De esta manera, la función soporta tanto la generación automática de datos como la configuración manual para escenarios de prueba más controlados.

5.6. Obtención de los Elementos y de la Capacidad

5.6.1. Obtención de Elementos de Forma Preestablecida

Esta funcionalidad proporciona los conjuntos de datos preestablecidos para validar los algoritmos, los cuales son los dados por el enunciado.

```
def obtener_elementos_predefinidos(factor_decision):
    if factor_decision == "v":
        capacidad_maxima = 4200
        elementos = [
            #[numero, volumen, precio]
            [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]
        ]
    else:
        capacidad_maxima = 3000
        elementos = [
            #[numero, peso, precio]
            [1, 1800, 72],
            [2, 600, 36],
            [3, 1200, 60]
        ]
    return elementos, capacidad_maxima
```

Código 8: Obtención de los elementos dados por el enunciado

Estos conjuntos están diseñados para proporcionar casos de prueba reproducibles y evaluar el comportamiento de los algoritmos en escenarios controlados.

5.6.2. Obtención de Elementos de Forma Aleatoria

Este procedimiento genera un conjunto de elementos con valores aleatorios dentro de parámetros controlados, los cuales se basan en los valores de los conjuntos dados por el enunciado del problema.

```
def obtener_elementos_aleatorios(factor_decision):
    cant_elem_limite = 20
    cap_mochila_max = 10000
    precio_max_elem = 200
    cantidad_elementos = random.randint(1, cant_elem_limite)
    capacidad_maxima = random.randint(1, cap_mochila_max)
```

```
7     elementos = generar_elementos(cantidad_elementos,
          capacidad_maxima, precio_max_elem, True, factor_decision
          )
8     return elementos, capacidad_maxima
```

Código 9: Obtención de los elementos de una manera aleatoria

Los siguientes rangos son los elegidos en base a los conjuntos de elementos predefinidos por el enunciado para la generación aleatoria de instancias del problema:

- **Cantidad de elementos**: Entre 1 y 20
- **Capacidad de mochila**: Entre 1 y 10,000 unidades
- **Precio máximo**: Hasta 200 unidades monetarias

Esta función permite evaluar los algoritmos en escenarios variables y probar su robustez ante diferentes configuraciones del problema de la mochila.

5.6.3. Obtención de Elementos por Consola

Este procedimiento permite al usuario definir completamente el problema interactivamente, pudiendo personalizar los elementos en función de sus necesidades.

```
1  def obtener_elementos_consola(factor_decision):
2      cant_elem_limite = 20
3      cap_mochila_max = 10000
4      precio_max_elem = 200
5      cantidad_elementos = pedir_entero("Cantidad de objetos
           disponibles: ", 1, cant_elem_limite)
6      if factor_decision == "v":
7          capacidad_maxima = pedir_entero("Volumen maximo de la
               mochila: ", 1, cap_mochila_max)
8      else:
9          capacidad_maxima = pedir_entero("Peso maximo de la
               mochila: ", 1, cap_mochila_max)
10     print()
11     elementos = generar_elementos(cantidad_elementos,
           capacidad_maxima, precio_max_elem, False,
           factor_decision)
12     input("Presione Enter para continuar...")
13     limpiar_pantalla()
14     return elementos, capacidad_maxima
```

Código 10: Obtención de los elementos mediante la consola

Este se basará para limitar el rango de los valores que ingrese el usuario en los mismos parámetros establecidos en el método de obtención anterior. Además, es el procedimiento que utilizará la función que permite ingresar y validar un número entero.

5.6.4. Coordinación de Obtención de Elementos

Esta subrutina actúa como un despachador que dirige el flujo según la elección del usuario, siendo el código del mismo el siguiente:

```
def obtener_elementos_y_capacidad(eleccion_elementos,
   factor_decision):
    if eleccion_elementos == "p":
        return obtener_elementos_predefinidos(factor_decision)
    elif eleccion_elementos == "a":
        return obtener_elementos_aleatorios(factor_decision)
    else:
        return obtener_elementos_consola(factor_decision)
```

Código 11: Elección del tipo de generación de elementos

5.7. Visualización de los Elementos Generados

Presenta los elementos disponibles, adaptando las columnas según el factor de decisión (volumen/-peso). Proporciona una visualización clara de todos los elementos con sus respectivas características antes de ejecutar los algoritmos de optimización.

```
def mostrar_elementos(elementos, factor_decision):
    print()
    print("Elementos disponibles:")
    if factor_decision == "v":
        print(f"{'Numero':<8} {'Volumen':<10} {'Precio':<8}")
    else:
        print(f"{'Numero':<8} {'Peso':<10} {'Precio':<8}")
    print("-" * 28)
    for elem in elementos:
        print(f"{elem[0]:<8} {elem[1]:<10} {elem[2]:<8}")
    print()
```

Código 12: Visualización de los Elementos Generados

5.8. Búsqueda Exhaustiva de la Solución Óptima

Dentro de la búsqueda exhaustiva, el programa se apoya en tres funciones que trabajan de forma conjunta para explorar todas las combinaciones posibles de elementos y encontrar la solución óptima. A continuación, se detallan en profundidad cada una de las funciones que conforman la búsqueda exhaustiva.

5.8.1. Generación de las Soluciones

Este procedimiento es la responsable de producir todas las combinaciones posibles de elementos, desde la combinación vacía hasta aquella que incluye todos los elementos disponibles. Utiliza la función `combinations` del módulo `itertools` para generar cada subconjunto, calculando para

cada uno la suma de las medidas y de los precios de cada elemento, así como su factibilidad respecto a la capacidad máxima de la mochila, pudiéndose esta última el peso o el volumen. Cada solución se guarda con su número de orden, la lista de identificadores de elementos, el total de medida, el total de valor (precio) y un indicador booleano de factibilidad.

```
def generar_soluciones(elementos, capacidad):
    todas_soluciones = []
    numero_solucion = 0
    for r in range(len(elementos) + 1):
        for combinacion in combinations(elementos, r):
            numero_solucion += 1
            elementos_nums = [elem[0] for elem in combinacion]
            medida_total = sum(elem[1] for elem in combinacion)
            valor_total = sum(elem[2] for elem in combinacion)
            es_factible = medida_total <= capacidad
            todas_soluciones.append([
                numero_solucion,
                elementos_nums,
                medida_total,
                valor_total,
                es_factible
            ])
    return todas_soluciones
```

Código 13: Implementación de la función que genera todas las soluciones, factibles o no

5.8.2. Generación de Gráfica

La visualización de resultados de la búsqueda exhaustiva está a cargo de `graficar_distribucion_combinaciones()`. Esta función clasifica las combinaciones según su factibilidad y la cantidad de elementos que contienen, y genera un gráfico de barras que muestra la distribución de soluciones factibles y no factibles. La solución óptima se resalta con un marcador especial en color dorado y se acompaña de un recuadro con su información principal. Además, se anotan en las barras los conteos correspondientes y se añade una cuadrícula para mejorar la legibilidad. El gráfico se guarda como archivo PNG y también se muestra en pantalla.

```
def graficar_distribucion_combinaciones(todas_soluciones,
   elementos, solucion_optima, titulo_archivo):
    max_elementos = len(elementos) + 1
    y_factibles = [0] * max_elementos
    y_no_factibles = [0] * max_elementos
    for solucion in todas_soluciones:
        num_elementos = len(solucion[1])
        es_factible = solucion[4]
        if es_factible:
            y_factibles[num_elementos] += 1
        else:
            y_no_factibles[num_elementos] += 1
```

```
    x = list(range(max_elementos))
    plt.figure(figsize=(12, 7))
    plt.bar(x, y_factibles, color='lightgreen', edgecolor='black', label='Soluciones Factibles')
    plt.bar(x, y_no_factibles, bottom=y_factibles, color='lightcoral', edgecolor='black', label='Soluciones No Factibles')
    num_elementos_optimo = len(solucion_optima[1])
    altura_marca = y_factibles[num_elementos_optimo] + y_no_factibles[num_elementos_optimo] + max(y_factibles + y_no_factibles) * 0.1
    plt.scatter(num_elementos_optimo, altura_marca, s=300, c='gold', marker='*', edgecolors='orange', linewidth=2, label=f'Solucion Optima (Valor: {solucion_optima[3]})', zorder=5)
    plt.annotate('', xy=(num_elementos_optimo, y_factibles[num_elementos_optimo] + y_no_factibles[num_elementos_optimo]), xytext=(num_elementos_optimo, altura_marca), arrowprops=dict(arrowstyle='->', color='orange', lw=2.5))
    info_text = f"Solucion Optima:\n\nCantidad de elementos: {num_elementos_optimo}\nValor: {solucion_optima[3]}\nElementos: {solucion_optima[1]}"
    plt.text(0.02, 0.98, info_text, transform=plt.gca().transAxes, bbox=dict(boxstyle="round,pad=0.5", edgecolor='orange', facecolor="gold", alpha=0.8), verticalalignment='top', fontsize=10, weight='bold')
    plt.title('Distribucion de Combinaciones por Cantidad de Elementos (Busqueda Exhaustiva)')
    plt.xlabel('Cantidad de Elementos por Combinacion')
    plt.ylabel('Cantidad de Combinaciones')
    plt.legend()
    for i in range(len(x)):
        if y_factibles[i] > 0:
            plt.text(i, y_factibles[i]/2, str(y_factibles[i]), ha='center', va='center', color='darkgreen')
        if y_no_factibles[i] > 0:
            plt.text(i, y_factibles[i] + y_no_factibles[i]/2, str(y_no_factibles[i]), ha='center', va='center', color='darkred')
    plt.xticks(x)
    plt.grid(axis='y', linestyle='--', alpha=0.7)
    plt.tight_layout()
    plt.savefig(titulo_archivo + '.png', dpi=300, bbox_inches='tight')
    plt.show()
```

Código 14: Implementación de la función que genera la gráfica de la búsqueda exhaustiva

5.8.3. Coordinación de la Búsqueda Exhaustiva

La función `busqueda_exhaustiva(elementos, capacidad)` implementa la búsqueda exhaustiva del siguiente modo:

- Inicia un cronómetro con `perf_counter()` para medir el tiempo de ejecución
- Genera todas las combinaciones posibles llamando a `generar_soluciones`
- Filtra las soluciones factibles (que no exceden la capacidad máxima)
- Identifica la solución de mayor valor usando `max` con clave en el valor total (tercer elemento)
- Calcula el tiempo transcurrido y retorna:
  - Solución óptima
  - Tiempo de demora
  - Lista completa de soluciones
  - Soluciones factibles

```
1 def busqueda_exhaustiva(elementos, capacidad):
2     tiempo_inicio_exh = perf_counter()
3     todas_soluciones = generar_soluciones(elementos, capacidad)
4     soluciones_factibles = [sol for sol in todas_soluciones if
          sol[4] == True]
5     solucion_optima = max(soluciones_factibles, key=lambda x: x
          [3]) if soluciones_factibles else None
6     tiempo_fin_exh = perf_counter()
7     demora_exh = tiempo_fin_exh - tiempo_inicio_exh
8     return solucion_optima, demora_exh, todas_soluciones,
          soluciones_factibles
```

Código 15: Implementación de la búsqueda exhaustiva

5.9. Búsqueda Heurística (Algoritmo Greedy)

En cuanto a la búsqueda heurística, el programa ofrece la función `algoritmo_greedy()`. Esta estrategia:

- Calcula para cada elemento su relación valor/medida
- Ordena los elementos de forma descendente según este cociente
- Recorre la lista seleccionando elementos que no superen la capacidad restante
- Acumula tanto la medida como el valor total de los elementos seleccionados

El resultado es una aproximación rápida a la solución óptima, junto con el tiempo de ejecución medido del mismo modo que en la búsqueda exhaustiva.

```python
def algoritmo_greedy(elementos:list[list], capacidad_max):
    tiempo_inicio_greedy = perf_counter()
    elementos_greedy = []
    for elem in elementos:
        numero, medida, valor = elem
        relacion = valor/medida
        elementos_greedy.append([numero, medida, valor,
           relacion])
    elementos_greedy.sort(key=lambda x: x[3], reverse=True)
    mochila = []
    sum_capacidad_mochila = 0
    sum_precio_mochila = 0
    capacidad_restante = capacidad_max
    for elem in elementos_greedy:
        numero, medida, valor, relacion = elem
        if medida <= capacidad_restante:
            sum_capacidad_mochila += medida
            sum_precio_mochila += valor
            capacidad_restante -= medida
            mochila.append(elem)

    tiempo_fin_greedy = perf_counter()
    demora_greedy = tiempo_fin_greedy - tiempo_inicio_greedy

    return mochila, sum_precio_mochila, sum_capacidad_mochila,
       demora_greedy
```

Código 16: Implementación de la búsqueda heurística

5.10. Muestreo de los Resultados de la Búsqueda Exhaustiva

Presenta de manera integral los resultados de la búsqueda exhaustiva incluyendo la generación del gráfico estadístico. Además, adapta la terminología y unidades según el factor de decisión, muestra la solución óptima encontrada, las métricas de tiempo, y proporciona estadísticas sobre la cantidad total de soluciones analizadas.

```python
def mostrar_resultados_exhaustiva(elementos, solucion_optima,
   demora, todas_soluciones, soluciones_factibles,
   factor_decision):
    tipo_busqueda = "Volumen" if factor_decision == "v" else "
       Peso"
    unidad = "cm^3" if factor_decision == "v" else "gramos"
    nombre_archivo = "busq_exh_vol_graf" if factor_decision ==
       "v" else "busq_exh_peso_graf"
    print(f"Busqueda Exhaustiva en Funcion del {tipo_busqueda}"
       )
```

```
6     graficar_distribucion_combinaciones(todas_soluciones,
          elementos, solucion_optima, nombre_archivo)
7     print(f"Precio maximo (Exhaustiva): $ {solucion_optima[3]}
          ; {tipo_busqueda}: {solucion_optima[2]} {unidad}")
8     print(f"Elementos en mochila (Exhaustiva): {solucion_optima
          [1]}")
9     print(f"Tiempo de demora (Exhaustiva): {demora:.6f}
          segundos")
10    print(f"Cantidad de soluciones: {len(todas_soluciones)}\
          nCantidad de soluciones factibles: {len(
          soluciones_factibles)}")
```

Código 17: Implementación del muestreo de los resultados la búsqueda exhaustiva

5.11. Muestreo de los Resultados del Algoritmo Greedy

Exhibe los resultados obtenidos mediante el algoritmo greedy de forma concisa. Incluye el valor total obtenido, la medida utilizada, los elementos seleccionados y el tiempo de procesamiento, permitiendo la comparación directa con los resultados de la búsqueda exhaustiva.

```
1 def mostrar_resultados_greedy(mochila_greedy, precio, medida,
      demora, factor_decision):
2     tipo_busqueda = "Volumen" if factor_decision == "v" else "
          Peso"
3     unidad = "cm^3" if factor_decision == "v" else "gramos"
4     print(f"Busqueda Heuristica (Algoritmo Greedy) en Funcion
          del {tipo_busqueda}")
5     print(f"Precio obtenido (Greedy): $ {precio} ; {
          tipo_busqueda}: {medida} {unidad}")
6     print(f"Elementos en mochila (Greedy): {[elem[0] for elem
          in mochila_greedy]}")
7     print(f"Tiempo de demora (Greedy): {demora:.6f} segundos")
```

Código 18: Implementación del muestreo de los resultados la búsqueda heurística

5.12. Programa Principal

El flujo principal del programa coordina todas las funciones para resolver el problema de la mochila mediante ambos enfoques. Procesa los argumentos de entrada, obtiene los elementos según la configuración especificada, ejecuta tanto la búsqueda exhaustiva como el algoritmo greedy, y presenta los resultados de ambos métodos para permitir la comparación de eficiencia y efectividad entre los diferentes enfoques algorítmicos.

```
1 #Programa principal
2 limpiar_pantalla()
3 eleccion_elementos, factor_decision = ingreso_argumentos()
4 elementos, capacidad_maxima = obtener_elementos_y_capacidad(
      eleccion_elementos, factor_decision)
```

```
5 mostrar_elementos(elementos, factor_decision)
6 mochila_exh, demora_exh, todas_soluciones, soluciones_factibles
      = busqueda_exhaustiva(elementos, capacidad_maxima)
7 mochila_greedy, precio, medida, demora_greedy =
     algoritmo_greedy(elementos, capacidad_maxima)
8 mostrar_resultados_exhaustiva(elementos, mochila_exh,
     demora_exh, todas_soluciones, soluciones_factibles,
     factor_decision)
9 print()
10 print("-" * 74)
11 print()
12 mostrar_resultados_greedy(mochila_greedy, precio, medida,
     demora_greedy, factor_decision)
13 print()
```

Código 19: Programa principal del código

6. Salidas por pantalla

En esta sección analizaremos los resultados obtenidos luego de haber ejecutado el programa mediante alguna de las dos sintaxis propuestas en la sección 2 (Metodología de Desarrollo). Esto debido a que al correr el programa de alguna de estas maneras se estaría utilizando alguno de los dos conjuntos de elementos propuestos en el enunciado del problema.

6.1. Escenario 1: Optimización por Volumen

6.1.1. Elementos Disponibles del Escenario 1

El programa presenta una tabla clara con 10 elementos donde se observan características muy variadas. El elemento 8 es muy pequeño (60 cm³) pero con un precio modesto, mientras que el elemento 6 tiene un volumen elevado en comparación a su precio.

```
Elementos disponibles:
Numero    Volumen     Precio
----------------------------
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
```

Código 20: Lista de elementos utilizada en los ejercicios 1 y 2

6.1.2. Resultados de la Búsqueda Exhaustiva del Escenario 1

La búsqueda exhaustiva encontró la solución óptima con un valor de $299 utilizando 3888 cm³ de los 4200 cm³ disponibles, logrando un aprovechamiento del 95.5 % de la capacidad. La combinación óptima incluye 8 elementos: [1, 2, 3, 5, 6, 7, 8, 10], excluyendo estratégicamente los elementos 4 y 9.

```
Busqueda Exhaustiva en Funcion del Volumen
Precio maximo (Exhaustiva): $ 299 ; Volumen: 3888 cm^3
Elementos en mochila (Exhaustiva): [1, 2, 3, 5, 6, 7, 8, 10]
Tiempo de demora (Exhaustiva): 0.001891 segundos
Cantidad de soluciones: 1024
Cantidad de soluciones factibles: 923
```

Código 21: Resultados obtenidos del ejercicio 1

6.1.3. Resultados de la Búsqueda Heurística (Greedy) del Escenario 1

El algoritmo greedy obtuvo un valor de $299 usando 3888 cm³, representando el 95.5 % de la capacidad disponible. La selección fue: [8, 1, 2, 3, 10, 7, 5, 6], mostrando una estrategia diferente basada en la relación valor/volumen de cada elemento.

```
Busqueda Heuristica (Algoritmo Greedy) en Funcion del Volumen
Precio obtenido (Greedy): $ 299 ; Volumen: 3888 cm^3
Elementos en mochila (Greedy): [8, 1, 2, 3, 10, 7, 5, 6]
Tiempo de demora (Greedy): 0.000011 segundos
```

Código 22: Resultados obtenidos del ejercicio 2

6.2. Escenario 2: Optimización por Peso

6.2.1. Elementos Disponibles del Escenario 2

Este escenario presenta un problema más simple con solo 3 elementos, donde se observa que el elemento 1 tiene el mayor peso y precio, el elemento 3 ofrece un balance intermedio, y el elemento 2 es el más liviano.

```
Elementos disponibles:
Numero   Peso    Precio
----------------------------
1        1800        72
2        600         36
3        1200        60
```

Código 23: Lista de elementos utilizada en el ejercicio 3

6.2.2. Resultados de la Búsqueda Exhaustiva del Escenario 2

La solución óptima logra $132 con 3000 gramos exactos, utilizando completamente la capacidad disponible. La combinación óptima [1, 3] representa la mejor selección posible, dejando de lado únicamente al elemento 2.

```
Busqueda Exhaustiva en Funcion del Peso
Precio maximo (Exhaustiva): $ 132 ; Peso: 3000 gramos
Elementos en mochila (Exhaustiva): [1, 3]
Tiempo de demora (Exhaustiva): 0.000071 segundos
Cantidad de soluciones: 8
Cantidad de soluciones factibles: 7
```

Código 24: Resultados obtenidos de la búsqueda exhaustiva del ejercicio 2

6.2.3. Resultados de la Búsqueda Heurística (Greedy) del Escenario 2

El algoritmo greedy seleccionó los elementos [2, 3], obteniendo $96 con 1800 gramos, utilizando solo el 60 % de la capacidad disponible. Esta diferencia significativa demuestra una limitación importante del enfoque heurístico en este caso particular.

```
Busqueda Heuristica (Algoritmo Greedy) en Funcion del Peso
Precio obtenido (Greedy): $ 96 ; Peso: 1800 gramos
Elementos en mochila (Greedy): [2, 3]
Tiempo de demora (Greedy): 0.000008 segundos
```

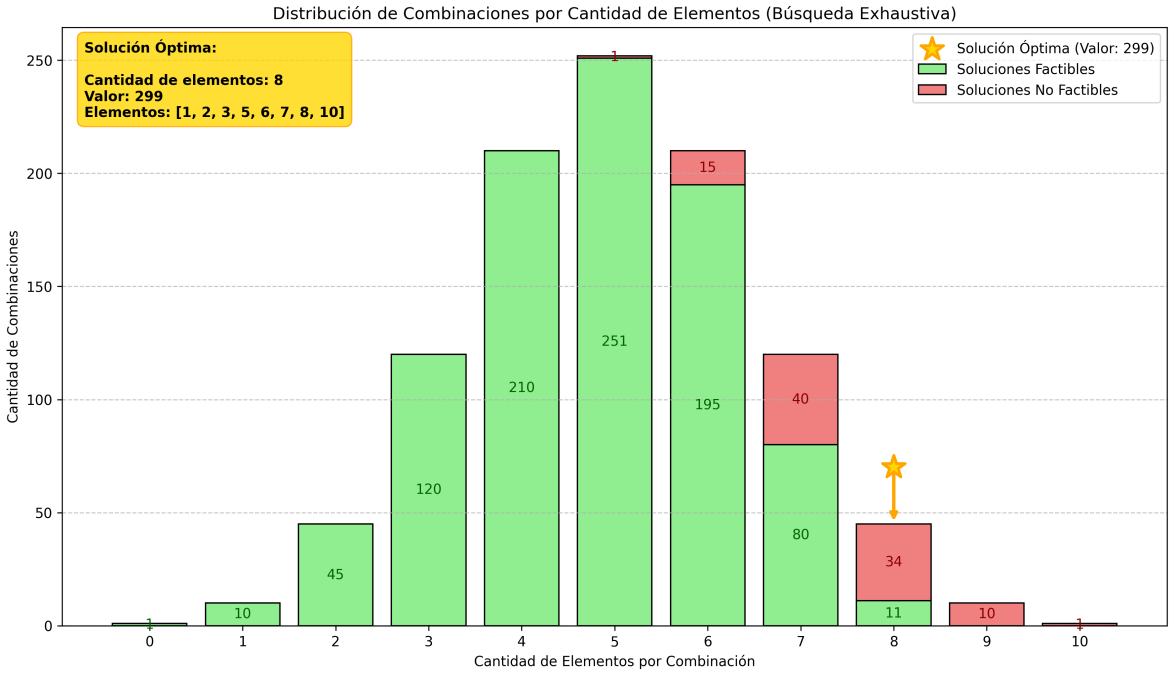
Código 25: Resultados obtenidos de la búsqueda exhaustiva del ejercicio 2

7. Gráficas de la Búsqueda Exhaustiva

Al igual que en la sección anterior se desarrollará lo obtenido en las gráficas en función de los dos conjuntos de elementos propuestos en el enunciado del problema.

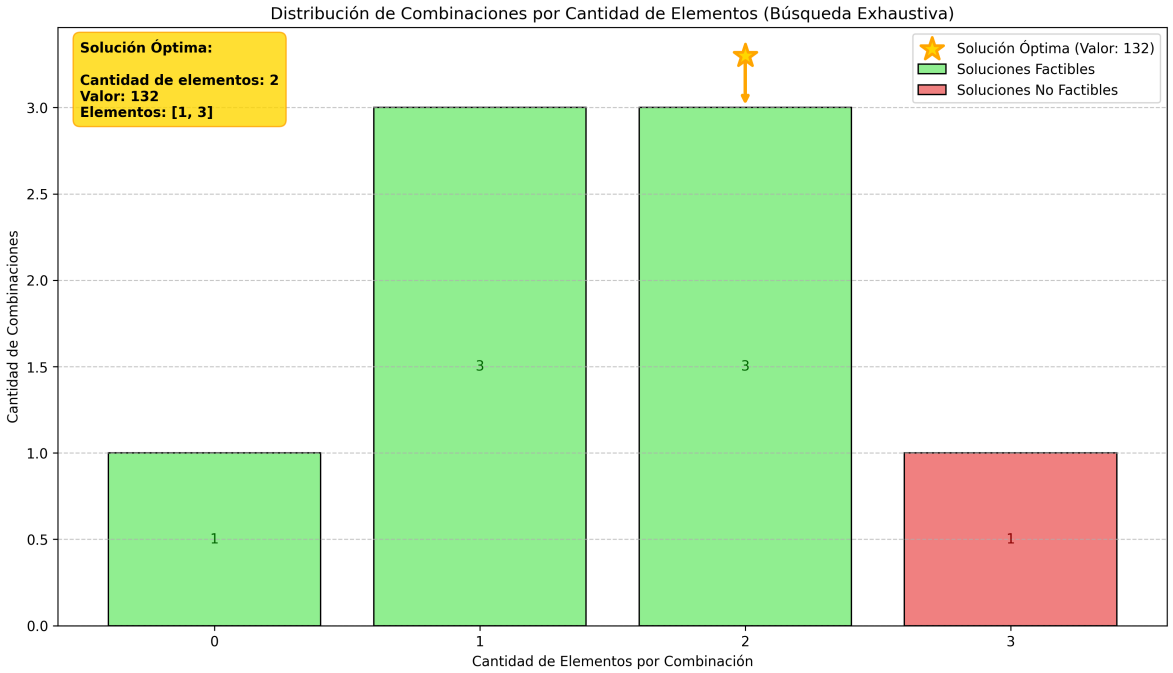
7.1. Gráfica de la Búsqueda Exhaustiva de la Optimización por Volumen

La figura presenta la distribución de las 1024 combinaciones posibles generadas mediante búsqueda exhaustiva para el conjunto de 10 elementos, considerando la restricción de volumen como el criterio de factibilidad. La estrella amarilla da a entender que la solución óptima obtenida está compuesta por 8 elementos, [1, 2, 3, 5, 6, 7, 8, 10], con un valor total de $299 y un volumen de 3888 cm^3. Se observa que las combinaciones con pocos elementos tienden a ser factibles, mientras que al aumentar la cantidad de elementos crece la proporción de soluciones no factibles, predominando en combinaciones de 8 o más elementos.



7.2. Gráfica de la Búsqueda Exhaustiva de la Optimización por Peso

La figura muestra la distribución de las 8 combinaciones posibles generadas mediante búsqueda exhaustiva para un conjunto de 3 elementos, considerando el peso como restricción de factibilidad. En este caso la estrella amarilla señala que la solución óptima obtenida está compuesta por 2 elementos, [1, 3], con un valor total de $132 y un peso de 3000 gramos. Se observa que, en este caso, la mayoría de las combinaciones son factibles, siendo la única combinación no factible la que incluye los tres elementos.



8. Conclusiones

Este trabajo práctico ha permitido analizar de manera integral el problema de la mochila mediante la implementación y comparación de dos enfoques algorítmicos fundamentales: la búsqueda exhaustiva y el algoritmo heurístico greedy. Los resultados obtenidos proporcionan información valiosa sobre el comportamiento, ventajas y limitaciones de cada método en diferentes escenarios.

8.1. ¿Eficiencia Computacional o Calidad de Solución?

En el escenario de optimización por volumen, la búsqueda exhaustiva garantizó encontrar una solución óptima de $299 de precio total evaluando las 1024 combinaciones posibles en 1.891 milisegundos. Por su parte, el algoritmo greedy logró exactamente la misma solución que la búsqueda exhaustiva, en tiempo considerablemente menor de apenas 0.011 milisegundos. Esta relación se modifica significativamente en el escenario de optimización por peso, donde el algoritmo greedy mostró una pérdida de efectividad más pronunciada, alcanzando solo el 72.7 % del valor óptimo, al obtener un precio de $96 contra un precio óptimo de $132). Este comportamiento ilustra que la efectividad del enfoque heurístico depende fuertemente de las características específicas del problema y la estructura de los datos.

8.2. Conclusión final

Los resultados demuestran que el algoritmo greedy es valioso para obtener soluciones de buena calidad en tiempos extremadamente reducidos, especialmente útil cuando el espacio de soluciones es demasiado grande para métodos exhaustivos. Mientras que evaluar 1024 combinaciones es factible en milisegundos, conjuntos de elementos mucho más grandes harían que se generaran millones de combinaciones, haciendo impracticable la búsqueda exhaustiva. Esto confirma que no existe un algoritmo superior por sobre otro para el problema de la mochila, sino que en su lugar la elección entre búsqueda exhaustiva y métodos heurísticos debe basarse en los requerimientos específicos del problema como la cantidad de soluciones, la eficiencia del algoritmo y la restricción del tiempo para obtener una solución al problema.