Parte 1

1. Análisis del enunciado: Un breve análisis del enunciado, identificando variables y dominio del problema

El problema se basa en el famoso problema de optimización de la mochila, pero con una adaptación. La adaptación consiste en que todos los ítems deben ser llevados, pero deben dividirse en la menor cantidad posible de mochilas (agentes). De esta manera, cada agente llevará un peso cercano al máximo y el peso se distribuirá de manera uniforme entre ellos. La mejor solución será aquella que requiera la menor cantidad de agentes para llevar todos los ítems. Las variables que deben considerarse son la cantidad de ítems en el input, su peso y la capacidad máxima de cada agente.

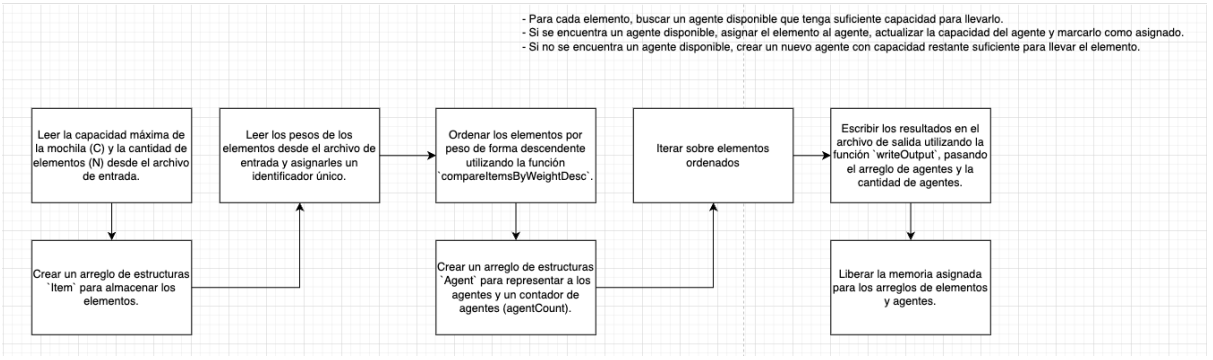
2. Planificación de solución: una idea a rasgos generales sobre cómo abordaron el problema, idealmente mencionando nombres de las funciones y structs a utilizar

La planificación de la solución se basa en un enfoque codicioso (greedy) para abordar el problema de optimización de la mochila adaptado. A continuación, se proporciona una descripción general de las funciones y estructuras utilizadas:

1. **Struct Item**: Representa un elemento individual que se puede colocar en la mochila. Contiene un identificador único (id) y un peso (weight).
2. **Struct Agent**: Representa a un agente encargado de llevar los elementos en la mochila. Contiene el número de agente (agent), la capacidad máxima de carga (capacity), un arreglo dinámico de índices de elementos (items) y la cantidad de elementos asignados al agente (item_count).
3. **Función compareItemsByWeightDesc**: Es una función de comparación utilizada por la función qsort para ordenar los elementos en base a su peso de forma descendente.
4. **Función greedyKnapsack**: Implementa el enfoque codicioso para resolver el problema. Recibe la capacidad máxima de la mochila (C), la cantidad de elementos (N), un arreglo de elementos (items) y un puntero al archivo de salida (output_stream). En esta función, los elementos se ordenan en base a su peso de forma descendente. Luego, se asignan los elementos a los agentes disponibles en función de su capacidad restante. Si no hay agentes disponibles, se crea uno nuevo. Los resultados se escriben en el archivo de salida llamando a la función writeOutput.
5. **Función writeOutput**: Escribe los resultados en el archivo de salida. Recibe el puntero al archivo de salida (output_stream), un arreglo de agentes (agents) y la cantidad de agentes (agentCount). La función imprime el número de agentes y, para cada agente, imprime los índices de los elementos asignados. Además, libera la memoria asignada para la lista de elementos de cada agente.

El código realiza la lectura del archivo de entrada, ejecuta la función greedyKnapsack y escribe los resultados en el archivo de salida.

3. Diagrama: Un diagrama/imagen donde indique gráficamente como se relacionan las distintas funciones y structs que implementaran/implementaron (Como una función llama a otra, que contiene el struct)


- Para cada elemento, buscar un agente disponible que tenga suficiente capacidad para llevarlo.
- Si se encuentra un agente disponible, asignar el elemento al agente, actualizar la capacidad del agente y marcarlo como asignado.
- Si no se encuentra un agente disponible, crear un nuevo agente con capacidad restante suficiente para llevar el elemento.
Leer la capacidad máxima de la mochila (C) y la cantidad de elementos (N) desde el archivo de entrada.
Crear un arreglo de estructuras `Item` para almacenar los elementos.
Leer los pesos de los elementos desde el archivo de entrada y asignarles un identificador único.
Ordenar los elementos por peso de forma descendente utilizando la función `compareItemsByWeightDesc`.
Crear un arreglo de estructuras `Agent` para representar a los agentes y un contador de agentes (agentCount).
Iterar sobre elementos ordenados
Escribir los resultados en el archivo de salida utilizando la función `writeOutput`, pasando el arreglo de agentes y la cantidad de agentes.
Liberar la memoria asignada para los arreglos de elementos y agentes.