Logotipo, nombre de la empresa

Descripción generada automáticamente

**Proyecto Final**

UVS Express – Distribución y Logística

**Alumnos**

Martin Gustavo Acosta Romero.

Raziel De Jesús Rodríguez Pérez.

**Docente**

Fleitas Toranzo Yadira

**Programa Educativo**

Ingeniería Informática.

Boca del Rio, Ver. A 11 de junio de 2025

Índice

[Algoritmos utilizados 3](#_Toc200513869)

[Precarga de datos 3](#_Toc200513870)

[Imprimir resultados productos 7](#_Toc200513871)

[Liberar tabla memorización 9](#_Toc200513872)

[Resolver Top Down 11](#_Toc200513873)

[Asignación optima productos camiones 14](#_Toc200513874)

[Inicialización de matrices de rutas 16](#_Toc200513875)

[Optimización de rutas (Modo 1: Dijkstra\_tiempo) 18](#_Toc200513876)

[Optimización de rutas (Modo 2: floyd\_distancia) 21](#_Toc200513877)

[Optimizar\_y\_asignar\_rutas 23](#_Toc200513878)

[Heapsort (Ordenacion por nombre - Aún no asignados) 25](#_Toc200513879)

[Merge Sort (Ordenación por Peso - No asignados) 29](#_Toc200513880)

[Counting Sort (Ordenación por Volumen - No asignados) 32](#_Toc200513881)

[Hash sort (Ordenación por nombre - asignados) 36](#_Toc200513883)

[Quick sort (lomuto) (Ordenación por peso - asignados) 39](#_Toc200513884)

[Radix sort (Ordenación por peso - asignados) 43](#_Toc200513885)

[Búsqueda de producto por nombre (Búsqueda Binaria Recursiva) 46](#_Toc200513886)

[Submenu ordenar productos 49](#_Toc200513887)

[Menu Principal 51](#_Toc200513888)

[Conclusión 53](#_Toc200513889)

# Algoritmos utilizados

## Precarga de datosTexto El contenido generado por IA puede ser incorrecto.

Texto

El contenido generado por IA puede ser incorrecto.

**Descripción de la función precargarDatos**

La función precargarDatos() tiene como objetivo inicializar automáticamente los datos base del sistema antes de que el usuario comience a interactuar con él. Se ejecuta una única vez al inicio del programa (desde main()), y genera valores aleatorios para poblar las estructuras centrales del proyecto logístico:

1. Productos (productosArreglo[])  
   Se generan hasta 100 productos, cada uno con un ID, un nombre tipo “Producto X”, precio, peso y volumen aleatorios dentro de rangos definidos.
2. Localidades (localidadesArreglo[])  
   Se crean hasta 20 localidades, con nombres, coordenadas geográficas (latitud y longitud) aleatorias y una cantidad aleatoria de pedidos (1 a 6).  
   Cada pedido incluye una referencia a un producto existente y una cantidad aleatoria.
3. Camiones (camionesArreglo[])  
   Se generan hasta 5 camiones con capacidades de carga y volumen aleatorias. Además, se asignan aleatoriamente un origen y destino (sin repetirse).
4. Clientes (clientesArreglo[])  
   Hasta 10 clientes con nombre, dirección ficticia y un historial de pedidos. Cada pedido en el historial contiene productos aleatorios.
5. Conexiones entre localidades (conexionesArreglo[])  
   Se establecen rutas entre localidades de dos formas:
   * Primero, conexiones mínimas entre localidades consecutivas para garantizar conectividad total del grafo.
   * Luego, se añaden conexiones aleatorias (hasta llegar a 30), evitando duplicadas o bucles.

Cada bloque utiliza funciones como rand() y sprintf() para generar datos diversos y coherentes. Al final, el sistema queda completamente inicializado con un entorno de prueba realista y funcional.

**Justificación**

La implementación de precargarDatos() es esencial en este proyecto por varias razones clave:

1. **Simulación autónoma y sin dependencias externas**  
   Permite ejecutar el programa sin necesidad de que el usuario cargue manualmente datos desde archivos, facilitando pruebas rápidas, demostraciones y validaciones.
2. **Entorno de prueba dinámico y variable**  
   Al usar valores aleatorios, se garantiza que cada ejecución proporcione un entorno diferente, ayudando a verificar la robustez del sistema frente a múltiples configuraciones posibles.
3. **Cobertura completa de las estructuras**  
   Se asegura que todas las estructuras críticas del sistema (productos, localidades, camiones, clientes y conexiones) estén pobladas, lo cual es indispensable para que las funciones posteriores (asignación, rutas, ordenamientos) funcionen correctamente.
4. **Facilita el desarrollo modular y la depuración**  
   Tener datos precargados permite a los desarrolladores **enfocarse en la lógica de algoritmos** (como programación dinámica o algoritmos de grafos) sin preocuparse por entradas vacías o inválidas.
5. **Reproducibilidad parcial y variedad de escenarios**  
   Aunque los datos son aleatorios, siguen un patrón estructurado que respeta la lógica del negocio. Esto permite obtener variedad de escenarios con comportamiento realista.

## Texto El contenido generado por IA puede ser incorrecto.Imprimir resultados productos

**Descripción de la función imprimir\_resultados\_productos**

La función imprimir\_resultados\_productos tiene como objetivo mostrar los resultados de la asignación de productos a un camión específico, indicando también la localidad de destino.

Al comienzo, se imprime el número del camión y el nombre de la localidad asociada. Luego, según el valor del parámetro deci, se comporta de la siguiente manera:

* Si deci == 0:  
  Se interpreta que no se pudieron asignar productos a ese camión. En este caso, se imprime el primer producto no asignado, indicando su nombre, cantidad, peso y volumen, junto con un contador total general de productos no asignados.
* Si deci == 1:  
  Se recorren todos los productos asignados exitosamente al camión. Para cada uno, se imprime su nombre, valor total, peso y volumen. Además, se acumula el valor total de todos los productos entregados y se imprime al final, junto con el total de productos asignados.
* Si totalSeleccionados == 0 y deci no se puede determinar:  
  Se indica que no hay productos para procesar, mostrando un mensaje claro al respecto.

Esta función utiliza impresión en consola (printf) para comunicar los resultados directamente al usuario, facilitando la comprensión de lo que ocurrió en el proceso de asignación.

**Justificación**

Esta función es clave para la trazabilidad y evaluación del sistema, por las siguientes razones:

1. Claridad informativa inmediata  
   Ofrece un resumen claro y legible de lo que ocurrió en la asignación de productos a cada camión. El usuario puede identificar rápidamente si un camión fue cargado correctamente o si hubo productos que no se pudieron incluir.
2. Diagnóstico en tiempo real  
   Es útil para detectar cuellos de botella: por ejemplo, si ciertos productos quedan fuera por volumen o peso, el operador puede decidir reasignar rutas, optimizar carga o ajustar los parámetros.
3. Apoyo a decisiones logísticas  
   Saber qué camiones transportan qué productos (y cuáles quedaron fuera) permite tomar decisiones inteligentes en simulaciones o situaciones reales.
4. Versatilidad para informes  
   El formato con valores acumulados (valor total de la carga, conteos, etc.) permite integrar esta salida en reportes automatizados o visualizaciones más avanzadas, si se desea escalar el sistema.
5. Facilidad de depuración y validación  
   Durante el desarrollo o prueba del sistema, esta función sirve como un punto de control que verifica la correcta ejecución de los algoritmos de asignación, mostrando resultados visibles al programador o usuario.

## Liberar tabla memorización

Texto

El contenido generado por IA puede ser incorrecto.

**Descripción de la función liberar\_tabla\_memoizacion**

La función liberar\_tabla\_memoizacion() tiene como propósito liberar la memoria dinámica previamente asignada a una tabla tridimensional de tipo float, que fue utilizada en el proceso de memorización dentro del algoritmo de programación dinámica implementado en asignacion\_optima\_productos\_camiones().

Esta tabla tridimensional tiene como dimensiones:

* n: número de productos o pedidos.
* pesoMax: límite de peso del camión.
* volumenMax: límite de volumen del camión.

La tabla guarda los resultados parciales (subproblemas) del algoritmo que maximiza el valor de los productos asignados sin superar la capacidad del camión. Una vez completado el cálculo y reconstruida la solución óptima, la memoria ya no es necesaria, por lo que esta función recorre toda la estructura y libera ordenadamente cada bloque de memoria:

1. Libera cada subarreglo de volumen.
2. Luego cada subarreglo de peso.
3. Finalmente, libera el arreglo principal.

**Justificación**

La inclusión de esta función es crítica para un sistema que utiliza estructuras dinámicas complejas como parte de su algoritmo de optimización:

1. Prevención de fugas de memoria (memory leaks)  
   En aplicaciones que utilizan estructuras dinámicas en múltiples ciclos o escenarios, no liberar memoria correctamente puede saturar el sistema, consumir recursos innecesarios o generar errores impredecibles.
2. Buenas prácticas de programación en C  
   El lenguaje C no incluye recolector de basura automático, por lo que el programador debe ser explícito en la gestión de memoria. Esta función cumple con esa responsabilidad de manera ordenada y segura.
3. Reutilización segura del algoritmo  
   Al liberar la tabla al final de cada uso, se permite que el algoritmo asignacion\_optima\_productos\_camiones() pueda ser ejecutado múltiples veces sin acumular residuos de memoria de ejecuciones anteriores.
4. Escalabilidad del sistema  
   En entornos donde el número de productos o la capacidad del camión puede crecer (por ejemplo, al simular distintos escenarios), liberar memoria correctamente garantiza que el sistema pueda operar con datasets grandes sin colapsar.
5. Separación de responsabilidades  
   Encapsular la liberación de memoria en una función separada mejora la legibilidad del código y permite su reutilización o modificación independiente del resto del algoritmo.

## Resolver Top Down

Texto

El contenido generado por IA puede ser incorrecto.

**Descripción de la función resolver\_top\_down**

La función resolver\_top\_down() implementa un algoritmo de programación dinámica con memorización (top-down) para resolver una versión extendida del problema de la mochila, en la que se consideran dos restricciones simultáneas: el peso máximo y el volumen máximo que puede transportar un camión.

Parámetros:

* Un arreglo de Pedido que contiene productos solicitados por una localidad.
* Un índice i que representa la posición actual en el arreglo.
* El peso y volumen restantes (peso\_restante, volumen\_restante) del camión.
* Una tabla tridimensional tabla que memoriza resultados previos.
* Una estructura DecisionProductos que guarda si un pedido fue tomado o no.

Proceso:

1. Caso base y validaciones: Si i < 0 o si las restricciones son negativas, retorna 0.
2. Verificación en la tabla: Si el subproblema ya fue resuelto, retorna el valor almacenado (evita recálculo).
3. Evaluación de opciones:
   * Sin tomar el pedido actual: llama recursivamente a la función con el siguiente índice.
   * Tomando el pedido actual (si cabe): añade su valor y descuenta su peso y volumen.
4. Comparación y decisión:
   * Se guarda en la tabla el mejor valor entre tomarlo o no.
   * Se registra la decisión en dp->decisiones[i].

Finalmente, retorna el valor máximo que se puede obtener desde el pedido i hacia atrás, respetando las restricciones físicas del camión.

**Justificación**

Esta función es el núcleo del módulo de asignación óptima de productos a camiones. Su implementación es fundamental por varias razones:

1. Resuelve un problema logístico realista  
   A diferencia del clásico problema de la mochila que considera solo peso, aquí se añade una segunda dimensión (volumen), lo que refleja con mayor fidelidad las restricciones de transporte en logística real.
2. Uso eficiente de la programación dinámica con memorización  
   La técnica top-down evita cálculos redundantes al reutilizar resultados previos guardados en la tabla. Esto reduce significativamente el tiempo de ejecución respecto a soluciones puramente recursivas.
3. Permite reconstruir la solución óptima  
   Gracias a la estructura DecisionProductos, no solo se obtiene el valor máximo transportado, sino también qué productos concretos deben ser asignados al camión, lo que es clave para tomar decisiones en la operación.
4. Adaptable y escalable  
   El diseño modular de esta función permite aplicarla a diferentes tamaños de pedidos o camiones, facilitando su reutilización si el sistema crece o cambia en el futuro.
5. Complementa la lógica del sistema UVS EXPRESS  
   Su integración con el resto del proyecto permite que la asignación de productos a camiones no sea aleatoria ni manual, sino que siga una estrategia optimizada matemáticamente, elevando la eficiencia global del sistema.

**Complejidad**

Sean:

* n: cantidad de pedidos (i desde 0 hasta n − 1)
* W: peso m´aximo (peso restante)
* V: volumen m´aximo (volumen restante)

Como es una función de programación dinámica con memorización (tipo mochila multidimensional), el número de subproblemas únicos es n · W · V y cada subproblema toma tiempo constante.

Peor caso:

Imagen de la pantalla de un celular con letras

El contenido generado por IA puede ser incorrecto.

Imagen que contiene objeto, reloj

El contenido generado por IA puede ser incorrecto.

Imagen que contiene objeto, antena, reloj

El contenido generado por IA puede ser incorrecto.



Mejor caso:



Imagen que contiene Texto

El contenido generado por IA puede ser incorrecto.



Caso promedio:



Texto

El contenido generado por IA puede ser incorrecto.



## Asignación optima productos camiones



**Descripción**

El algoritmo implementado para asignar productos a camiones de forma óptima está basado en programación dinámica con memoización, una técnica eficiente que permite resolver problemas complejos dividiéndolos en subproblemas más simples, cuyos resultados se almacenan para evitar cálculos repetitivos.

En esencia, el algoritmo resuelve una variación del problema clásico de la mochila (Knapsack Problem). Mientras que en el problema clásico solo se considera una restricción (el peso máximo que puede soportar la mochila), en nuestro caso se incorporan dos restricciones simultáneas: el peso y el volumen que cada camión puede transportar. Esto convierte el problema en una versión más compleja conocida como “Mochila multidimensional” o “Mochila con múltiples restricciones”.

Durante la ejecución del algoritmo, se analiza cada pedido (producto) y se evalúan dos decisiones posibles:

1. No incluir el producto en el camión, dejando los recursos disponibles (peso y volumen) intactos.
2. Incluir el producto, si su peso y volumen no exceden los límites actuales del camión, sumando su valor al total y actualizando los recursos restantes.

Mediante la memoización, cada combinación de índice de producto, peso restante y volumen restante se guarda en una tabla para que, si se vuelve a necesitar, el resultado se recupere directamente sin recalcularse.

**Justificación**

Elegimos esta técnica porque:

* Maximiza la eficiencia frente a una exploración total (fuerza bruta), que sería exponencial en tiempo.
* Evita recomputaciones, lo cual es esencial en problemas con múltiples subproblemas superpuestos, como ocurre al considerar todas las combinaciones posibles de productos, peso y volumen.
* Produce soluciones óptimas garantizadas, ya que se exploran todas las decisiones posibles de manera estructurada.

Frente a otras técnicas como algoritmos voraces (greedy) o búsqueda local, la programación dinámica:

* No se conforma con soluciones subóptimas,
* Y es preferible cuando se necesita precisión en la asignación, especialmente en sistemas logísticos donde cada recurso cuenta.

## Inicialización de matrices de rutas



**Descripción**

La función inicializar\_matrices\_rutas() cumple un rol fundamental en el sistema de logística al construir y llenar dos matrices clave: una matriz de tiempo estimado de viaje (matriz\_tiempo) y una matriz de distancia geográfica (matriz\_distancia). Ambas matrices representan la red de conexiones entre localidades dentro del sistema.

Inicialmente, las matrices se llenan con valores infinitos (∞), excepto en las diagonales principales que se asignan a cero, representando que el tiempo/distancia de una localidad consigo misma es nulo. Luego, se recorren todas las conexiones disponibles (conexionesArreglo[]) para actualizar estas matrices con el menor tiempo y la menor distancia posible entre pares de localidades conectadas directamente.

El tiempo de viaje se calcula sumando el tiempo base (tiempo\_min) con una posible penalización por tráfico (penalizacion\_trafico). Estos valores actualizan las matrices solo si representan una mejora respecto a los valores previamente almacenados.

**Justificación**

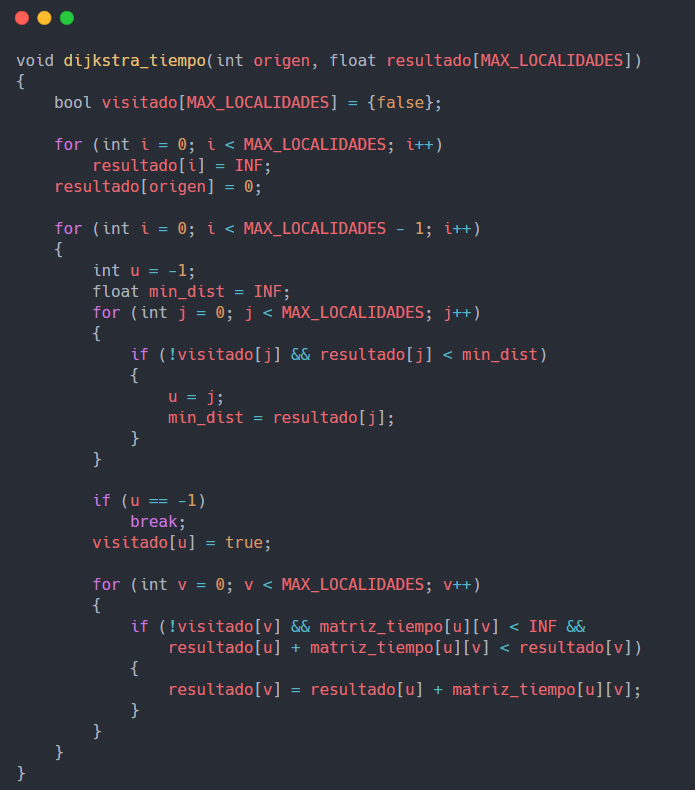
Esta función es esencial porque prepara los datos sobre los cuales operan los algoritmos de rutas óptimas:

* Dijkstra utiliza matriz\_tiempo para hallar el camino más rápido entre localidades, lo cual es vital cuando se desea minimizar el tiempo de entrega.
* Floyd-Warshall emplea matriz\_distancia para calcular todas las rutas más cortas en términos de distancia, útil para minimizar el consumo de combustible u optimizar rutas por proximidad.

Si estas matrices no se construyen correctamente:

* No habría forma eficiente de determinar si una localidad está conectada a otra.
* Se podrían obtener resultados erróneos o rutas inexistentes en los cálculos de optimización.
* Los algoritmos de rutas trabajarían con datos incompletos o incorrectos, afectando directamente la eficiencia logística.

## Optimización de rutas (Modo 1: Dijkstra\_tiempo)



**Descripción:**

El algoritmo dijkstra\_tiempo implementa el clásico algoritmo de Dijkstra para encontrar las rutas más rápidas entre una localidad de origen y todas las demás, utilizando como peso el tiempo total de recorrido (minutos), considerando también una penalización por tráfico. Se basa en una estructura de matriz de adyacencia llamada matriz\_tiempo que contiene los valores de tiempo entre pares de localidades.

La función recorre todos los nodos (localidades) y, usando un arreglo de visitados y otro de distancias, encuentra el camino de menor tiempo acumulado desde el origen a cada destino. En cada iteración selecciona el nodo no visitado con menor tiempo acumulado y actualiza las distancias de sus vecinos.

**Justificación del uso del algoritmo Dijkstra**

El algoritmo de Dijkstra es ideal para resolver el problema de rutas más rápidas (modo tiempo) porque:

* Precisión: Encuentra la ruta óptima en términos de menor tiempo total sin ciclos negativos.
* Eficiencia en grafos densos de tamaño moderado: Como el sistema considera hasta 20 localidades, Dijkstra tiene buen rendimiento con complejidad O(n^2) usando arreglos simples, o O((n+e)logn) con colas de prioridad.
* Adaptabilidad: Puede ajustarse fácilmente para considerar penalizaciones por tráfico sumando valores al tiempo base.

**Comparación con otras opciones**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algoritmo** | **Complejidad** | **Soporta pesos negativos** | **Ventajas** | **Desventajas** |
| Dijkstra | O(n^2) o O((n + e)\log n) | No | Eficiente, simple, resultados óptimos en grafos no negativos | No útil si se necesita manejar penalizaciones negativas |
| Bellman-Ford | O(n⋅e) | Sí | Maneja pesos negativos | Más lento en grafos densos |
| A\* | O(n) en el mejor caso | No | Eficiente con heurísticas | Requiere función heurística adecuada |
| Floyd-Warshall | O(n^3) | Sí | Todos los caminos entre todos los nodos | Costoso en tiempo y espacio para grandes grafos |

**Análisis de complejidad**

Sea:

* L = MAX LOCALIDADES (número de localidades)

La estructura es:

Esquemático

El contenido generado por IA puede ser incorrecto.

Resolviendo la sumatoria:

*T(L) = t1 + t2L + (L − 1) [t3L + t4 + t5L]*

*T(L) = t1 + t2L + (L − 1)t3L + (L − 1)t4 + (L − 1)t5L*

*T(L) = t1 + t2L + (t3 + t5)L(L − 1) + (L − 1)t4*

Redefiniendo *c1 = t3 + t5, c2 = t1 + t2L + (L − 1)t4:*

*T(L) = c1L(L − 1) + c2*

*T(L) = c1(L 2 − L) + c2*

**Caso promedio:**

Para cotas asintóticas, el termino dominante es L^2

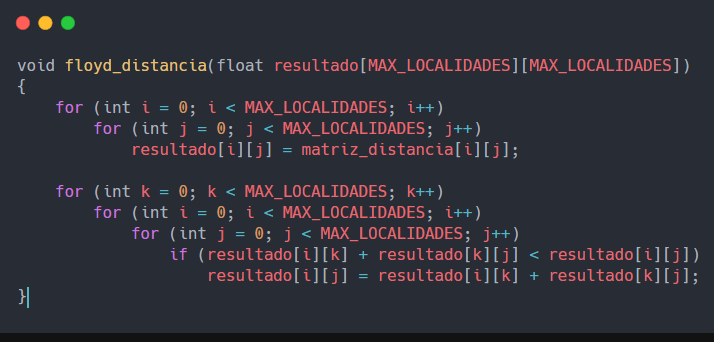
*c1 L^2*

Por lo tanto:

Texto, Carta

El contenido generado por IA puede ser incorrecto.

## Optimización de rutas (Modo 2: floyd\_distancia)

****

**Descripción:**

El algoritmo floyd\_distancia implementa el algoritmo clásico de Floyd-Warshall, diseñado para encontrar las distancias mínimas entre todas las parejas de localidades en la red. Se basa en programación dinámica y utiliza una matriz de adyacencia (matriz\_distancia) que representa las distancias directas entre localidades. Posteriormente, esta matriz es actualizada iterativamente para reflejar la menor distancia posible entre cada par de nodos, considerando caminos intermedios.

Se ejecutan tres bucles anidados de tamaño V, donde V es el número de localidades. El algoritmo actualiza la distancia entre cada par (i,j) si pasar por un nodo intermedio k representa una mejora.

**Justificación del uso de Floyd-Warshall**

Aunque su complejidad es O(V^3), el algoritmo es óptimo para grafos pequeños y densos, como el del proyecto (20 localidades). Además:

* Permite calcular todas las distancias mínimas en una sola ejecución.
* A diferencia de Dijkstra, no necesita estructuras adicionales como colas de prioridad.
* Es estable y simple de implementar en C con matrices estáticas.
* Recomendado por la profesora, cumple con el enfoque académico esperado.

**Comparación con otras opciones**

|  |  |  |  |
| --- | --- | --- | --- |
| **Algoritmo** | **Complejidad** | **Ventajas** | **Desventajas** |
| Floyd-Warshall | O(V^3) | Simple, útil para todos-a-todos, estable | No escala bien a grafos grandes |
| Dijkstra (repetido) | O(V^3) sin heap | Más preciso si se personaliza por nodo | Requiere múltiples ejecuciones |
| A\* o heurísticos | Depende de heurística | Más rápido en algunos casos reales | No garantiza mínima distancia total |

**Análisis de complejidad**

Sea:

* L = MAX LOCALIDADES (número de localidades)

Con los tres ciclos anidados para el algoritmo de Floyd-Warshall

for k = 0 to L−1 : for i = 0 to L−1 : for j = 0 to L−1 : if (...): resultado[i][j]

La estructura es:

T(L) = t1L^2 + t2L^3

**Caso promedio:**

Para cotas asintóticas, el termino dominante es t2L^3

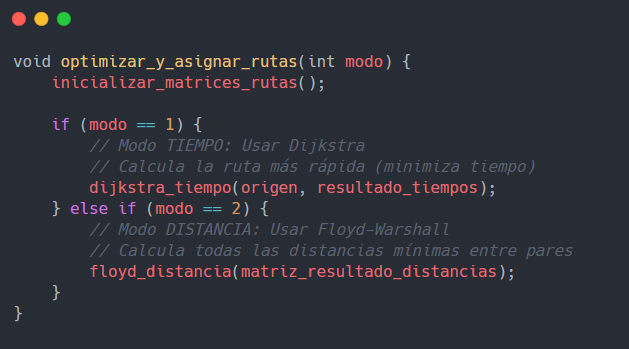
Por lo tanto:

Texto, Carta

El contenido generado por IA puede ser incorrecto.

## Optimizar\_y\_asignar\_rutas

(Imagen sintetizada de toda la función)



**Descripción:**

La función optimizar\_y\_asignar\_rutas se encarga de calcular y mostrar la ruta más eficiente que debe seguir cada camión para realizar sus entregas, según uno de dos criterios de optimización: tiempo o distancia.

Proceso:

1. Inicializa las matrices de rutas (matriz\_tiempo y matriz\_distancia), que representan los pesos (tiempo y distancia) entre las localidades conectadas por carreteras.
2. Recorre cada camión disponible del sistema.
3. Para cada camión, se identifica su localidad de origen y destino.
4. Dependiendo del valor del parámetro modo, se ejecuta uno de los dos algoritmos:
   * Modo 1 – Optimización por tiempo:  
     Se utiliza el algoritmo de Dijkstra, que calcula el tiempo mínimo desde el origen hasta todas las localidades, y se selecciona el tiempo hasta el destino del camión. Este cálculo considera penalizaciones por tráfico.
   * Modo 2 – Optimización por distancia:  
     Se utiliza el algoritmo de Floyd-Warshall, que calcula todas las distancias mínimas entre cada par de localidades. Se toma la distancia entre el origen y el destino del camión como la ruta más corta.
5. Finalmente, se imprime una tabla con los resultados: ID del camión, localidad de origen, localidad de destino y el tiempo o distancia total estimada de la ruta.

**Justificación**

Decidimos implementar la funcionalidad de optimizar y asignar rutas dentro de una única función modular por las siguientes razones técnicas y estructurales:

**1. Claridad y separación de responsabilidades**

Agrupar la lógica de optimización y asignación en una sola función permite separar claramente esta parte del sistema de otras funcionalidades como la carga de productos, la gestión de clientes o la visualización de resultados. Esto mejora la legibilidad del código y facilita su mantenimiento.

**2. Centralización del flujo de decisión**

Diseñar una función que centralice la elección de la mejor ruta para cada camión permite manejar de forma ordenada la información relacionada con:

* Origen y destino de cada camión.
* Información de la red de carreteras (tiempo, distancia, penalizaciones).
* Registro de la ruta asignada y su impresión.

Esta centralización evita redundancia de código y asegura que todas las decisiones de ruta sigan el mismo formato y estructura.

**3. Escalabilidad**

El enfoque modular hace que sea sencillo extender la funcionalidad en el futuro. Por ejemplo, si se desean agregar nuevas métricas de optimización (costos, emisiones, riesgo), se puede adaptar la lógica interna sin modificar otras partes del sistema.

**4. Cumplimiento con los objetivos del proyecto**

El proyecto exige que se asignen rutas optimizadas por criterios específicos. Nuestra función permite automatizar ese proceso, respetando restricciones como:

* Solo considerar camiones disponibles.
* Validar rutas existentes entre localidades.
* Calcular el valor óptimo de cada ruta de manera eficiente y presentarlo de forma clara.

**5. Facilidad para pruebas y validación**

Al tener toda la lógica de optimización de rutas en un solo lugar, es más fácil:

* Ejecutar pruebas de escritorio.
* Depurar errores en los recorridos.
* Comparar resultados entre distintas estrategias de optimización.

## Heapsort (Ordenacion por nombre - Aún no asignados)

**Descripción:**

El algoritmo de Heapsort implementado en el sistema se utiliza para ordenar alfabéticamente los productos no asignados (no\_seleccionados) según su nombre. La estructura base del algoritmo es un montículo mínimo (min-heap), construido sobre un arreglo dinámico que almacena objetos del tipo ProductoSeleccionado.

El ordenamiento ocurre en dos fases:

1. Construcción del heap: Cada producto se inserta en el heap con una operación push\_heap, que reorganiza los elementos para mantener la propiedad de orden (nombre menor en la raíz). Esta inserción tiene complejidad O(log n).
2. Extracción ordenada: Se extraen los productos uno a uno con pop\_heap, obteniendo en cada iteración el producto con el nombre lexicográficamente menor. Esta extracción también toma O(log n).

Así, el algoritmo final heap\_sort\_nombres realiza ambas fases para obtener un arreglo ordenado con complejidad total de O(n log n).

**Justificación del uso de Heapsort**

El uso de Heapsort está plenamente justificado para esta funcionalidad del sistema por las siguientes razones:

**1. Rendimiento garantizado**

* Heapsort ofrece una complejidad O(n log n) en el peor caso, a diferencia de Quicksort que puede degradarse a O(n²) si el pivote se elige mal.
* Esta característica lo hace adecuado para situaciones donde se necesita un rendimiento consistente sin depender de la distribución inicial de los datos.

**2. Ordenamiento personalizado**

* La implementación permite comparar productos por su nombre utilizando strcmp, lo cual es ideal para ordenar cadenas de texto y adaptar el criterio de orden fácilmente.
* Esto ofrece mayor control que funciones genéricas como qsort.

**3. Simplicidad y control de memoria**

* Aunque Heapsort no es un algoritmo "estable" (no conserva el orden de elementos con claves iguales), esto no representa una desventaja aquí, ya que el campo nombre es único por producto.
* El uso de un heap explícito permite una implementación intuitiva y controlada de la lógica de comparación.

**4. Adecuado para tamaño medio**

* Dado que MAX\_PRODUCTOS es 100, el uso de Heapsort es razonable en términos de consumo de recursos, manteniendo el balance entre eficiencia y claridad en la lógica.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algoritmo** | **Complejidad** | **Estable** | **Ventajas** | **¿Por qué no se eligió?** |
| **Heapsort** | O(n log n) | ❌ | Consistencia, control | Se eligió — buena opción en este caso |
| Quicksort | O(n log n) / O(n²) | ❌ | Muy rápido en promedio | Menos confiable en el peor caso |
| Mergesort | O(n log n) | ✅ | Estabilidad | Requiere más memoria (O(n)) |
| Counting/Radix | O(n + k) / O(nk) | ✅ | Muy rápido con enteros | No aplica bien para strings |
| qsort estándar | Depende | Depende | Simplicidad en C | Menos control sobre el comportamiento |

**Comparación con otros algoritmos**

**Complejidad**

Las operaciones principales en heap sort nombres son:

* Inicialización del heap: O(n)
* Insertar n elementos en el heap (push heap): cada inserción tiene complejidad O(log n), por lo tanto: O(n log n)
* Extraer n elementos en orden (pop heap): cada extracción tiene complejidad O(log n), por lo tanto: O(n log n)

Sumando todas las operaciones relevantes:

*T(n) = O(n) + O(n log n) + O(n log n) = O(n log n)*

Mejor, peor y caso promedio:

*T(n) = O(n log n)*

Calculo:

Texto, Carta

El contenido generado por IA puede ser incorrecto.

*log((n − 1)!) ≈ (n − 1) log(n − 1) − (n − 1)*

*Tp(n) ≈ t123 + (n − 1)t3458 + t567 · ((n − 1) log(n − 1))*

*Tp(n) = c1n log n + c2n + c3*

La complejidad temporal del algoritmo Heap Sort es:

*O(n log n)*

## Merge Sort (Ordenación por Peso - No asignados)



**Descripción**

Para ordenar los productos no asignados por peso total en mi sistema de logística, decidimos implementar el algoritmo Merge Sort. Esta función la llamo directamente desde el submenú de ordenamiento cuando elijo la opción de “Ordenar por peso (Aun no asignados)”.

Lo que hace Merge Sort en nuestro código es tomar el arreglo no\_seleccionados[], que contiene los productos que no pudieron ser cargados en los camiones, y lo ordena de forma ascendente según el campo peso\_total. La idea es que, al tenerlos ordenados, pueda analizar mejor qué productos podrían intentar asignarse en una próxima iteración o simplemente visualizarlos con más claridad.

**Justificación del uso de Merge Sort**

* Estabilidad: Merge Sort es estable, lo que significa que mantiene el orden relativo de elementos iguales, útil si otros atributos del producto se deben conservar.
* Eficiencia garantizada: Tiene un tiempo de ejecución de O(n log n) en el peor caso, ideal cuando necesitas rendimiento consistente.
* No depende de los valores del arreglo (a diferencia de Counting o Radix Sort), por lo que es adecuado para ordenar por campos como peso\_total que son valores decimales.

**Comparación con otras alternativas**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algoritmo** | **Complejidad (Peor Caso)** | **Estable** | **Uso en tu código** | **¿Adecuado para peso?** |
| **Merge Sort** | O(n log n) | ✅ | Ordenar por peso no asignado | ✅ Sí |
| Quick Sort (Lomuto) | O(n²) (prom: n log n) | ❌ | Ordenar por peso asignado | ⚠️ Puede ser peor |
| Heap Sort | O(n log n) | ❌ | Ordenar por nombre no asignado | ❌ (menos legible) |
| Counting Sort | O(n + k) | ✅ | Ordenar por volumen (si valores enteros) | ❌ (peso no aplica) |
| Radix Sort | O(n·k) | ✅ | Ordenar por volumen (usando enteros) | ❌ (peso decimal) |

**Complejidad**

La recurrencia del algoritmo es:

Texto

El contenido generado por IA puede ser incorrecto.

Expandimos varias veces:

Texto, Carta

El contenido generado por IA puede ser incorrecto.

Después de k=3 iteraciones llegamos a la conclusión que:

Imagen que contiene Texto

El contenido generado por IA puede ser incorrecto.

El proceso termina cuando n/2^k = 1 y k = log2 n:

*T(n) = nT(1) + cn log2 n*

*T(n) = Θ(n log n)*

Complejidad en todos los casos:

*Θ(n log n)*

## Counting Sort (Ordenación por Volumen - No asignados)

## Texto El contenido generado por IA puede ser incorrecto.

**Descripción:**

En esta implementación, utilizo el algoritmo Counting Sort para ordenar los productos *no asignados* según su volumen total. Dado que el volumen puede tener decimales, lo multiplico por 100 para convertirlo a un número entero antes de procesarlo. Counting Sort funciona contando la cantidad de ocurrencias de cada valor y luego acumulando dichas cuentas para determinar la posición final de cada elemento ordenado.

El proceso general es el siguiente:

1. Calculo el valor máximo del volumen (convertido a entero) entre todos los productos.
2. Creo un arreglo count[] para contar la frecuencia de cada volumen.
3. Transformo count[] en una tabla acumulativa.
4. Uso la tabla acumulativa para ubicar cada producto en su posición ordenada dentro de un arreglo salida[].

**Justificación:**

Elegimos Counting Sort porque se adapta muy bien a datos con rango limitado y discretos como lo es el volumen multiplicado por 100. Además, es un algoritmo lineal en tiempo para este caso particular (cuando el rango de valores no es muy grande), lo que lo hace más rápido que algoritmos comparativos como Quick Sort o Merge Sort.

En el contexto del sistema, los productos no asignados son relativamente pocos (máximo 100) y sus volúmenes también están acotados, por lo que Counting Sort es extremadamente eficiente y predecible para este propósito.

**Comparación con otras opciones**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algoritmo** | **Complejidad promedio** | **Estable** | **Adecuado para...** | **Comparación con Counting Sort** |
| **Counting Sort** | O(n + k) | Sí | Valores enteros pequeños o rango acotado | Más rápido cuando los datos tienen poco rango |
| **Quick Sort** | O(n log n) | No | Datos generales grandes | Más versátil pero inestable y peor rendimiento en casos desfavorables |
| **Merge Sort** | O(n log n) | Sí | Cualquier tipo de datos | Más estable, pero con mayor uso de memoria |
| **Radix Sort** | O(nk) | Sí | Claves de longitud fija/dígitos | Similar eficiencia, pero más complejo de implementar |

**Análisis de complejidad**

Sea:

* n = número de elementos del arreglo arr
* k = valor máximo entero de volumen total \* 100 (guardado como max val)

Búsqueda de max val: ciclo de i = 1 a n − 1 (t1 por iteración)

Un reloj en el medio

El contenido generado por IA puede ser incorrecto.

Inicialización del arreglo count [0..k]: ciclo de i = 0 a k (t2 por iteración)

Imagen que contiene objeto, reloj

El contenido generado por IA puede ser incorrecto.

Conteo de frecuencias: ciclo de i = 0 a n − 1 (t3 por iteración)

Imagen de la pantalla de un celular con letras

El contenido generado por IA puede ser incorrecto.

Suma acumulada de frecuencias: ciclo de i = 1 a k (t4 por iteración)

Imagen que contiene objeto, reloj

El contenido generado por IA puede ser incorrecto.

Reordenamiento: ciclo de i = n − 1 a 0 (t5 por iteración)

Esquemático

El contenido generado por IA puede ser incorrecto.

Copia del arreglo ordenado: ciclo de i = 0 a n − 1 (t6 por iteración)

Un reloj en el medio

El contenido generado por IA puede ser incorrecto.

Sumando todos los términos:

*T(n, k) = t0 + (n − 1)t1 + (k + 1)t2 + nt3 + kt4 + nt5 + nt6*

Agrupando términos:

*T(n, k) = c1n + c2k + c3*

Donde:

*c1 = t1 + t3 + t5 + t6, c2 = t2 + t4 y c3 = t0 − t1 + t2*

**Caso promedio:**

En general, el algoritmo tiene costo proporcional a n + k

Tpr(n, k) = c1n + c2k + c3

Tpr(n, k) ∈ Θ(n + k)

Texto, Carta

El contenido generado por IA puede ser incorrecto.

## Hash sort (Ordenación por nombre - asignados)



**Descripción:**

Este algoritmo implementa una combinación de tabla hash y ordenamiento por nombre. El objetivo es organizar los productos asignados (si\_seleccionados) alfabéticamente por nombre, agrupándolos eficientemente mediante una estructura de dispersión.

Funcionamiento:

1. Inicialización de la tabla hash: Se limpia cada bucket de la tabla tablaHash[HASH\_SIZE].
2. Hashing por primera letra: Se calcula el índice usando la función hash\_nombre, que retorna el valor ASCII de la primera letra del nombre módulo el tamaño de la tabla.
3. Inserción en la tabla hash: Cada ProductoSeleccionado se inserta en su bucket correspondiente como una lista enlazada.
4. Ordenamiento dentro de cada bucket: Se extraen los elementos del bucket a un arreglo temporal y se ordenan usando qsort y la función comparar\_nombre.
5. Impresión en orden alfabético: Se imprimen los productos ordenados lexicográficamente por nombre dentro de cada bucket.

**Justificación del uso de Hash sort**

* Se trata de una técnica eficiente cuando los datos están distribuidos homogéneamente.
* Al reducir la cantidad de elementos a ordenar por bucket, se mejora el rendimiento del ordenamiento al minimizar las llamadas a qsort.
* Ideal para casos donde hay grupos definidos por letras iniciales o patrones comunes en nombres, como "Producto 1", "Producto 2", etc.
* Permite mantener una estructura accesible y escalable para visualizar productos agrupados y ordenados.

**Comparación con otros algoritmos**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algoritmo** | **Complejidad (promedio)** | **Estable** | **Mejor uso** | **Comparación con hash\_sort\_asignados\_nombre** |
| hash\_sort\_asignados\_nombre | O(n) hashing + O(k log k) por bucket | Sí | Agrupación + orden por nombre | Más eficiente si hay muchos productos con letras iniciales distintas |
| heap\_sort\_nombres | O(n log n) | No | Orden global sin agrupación | No agrupa por inicial; ordena todo de forma global |
| quick\_sort\_lomuto\_peso | O(n log n) promedio | No | Orden por peso | Más rápido en promedio que heap, pero menos robusto con datos similares |
| mergeSort | O(n log n) | Sí | Orden por peso | Estable pero no agrupa ni indexa |
| counting\_sort\_volumen | O(n + k) | Sí | Valores discretos y acotados | Solo aplicable si el dominio de los datos es reducido |

**Complejidad**

Sea n el número total de productos asignados, y m = HASH SIZE el tamaño de la tabla hash.

Para hash sort asignados nombre:

Las principales operaciones son:

* Inicializar la tabla hash: O(m)
* Insertar n productos en la tabla hash: O(n)
* Para cada bucket (lista), recorrer sus elementos y ordenarlos lexicogr´aficamente: Si ki es la cantidad de productos en el bucket i,

Texto

El contenido generado por IA puede ser incorrecto.

* Imprimir los elementos: O(n)

La suma de todos los elementos en los buckets es Imagen que contiene objeto, reloj, medidor

El contenido generado por IA puede ser incorrecto., por lo tanto:

Gráfico, Gráfico de cajas y bigotes

El contenido generado por IA puede ser incorrecto.

En el peor de los casos (todos los productos caen en un solo bucket):

Tpeor(n) = O(n log n)

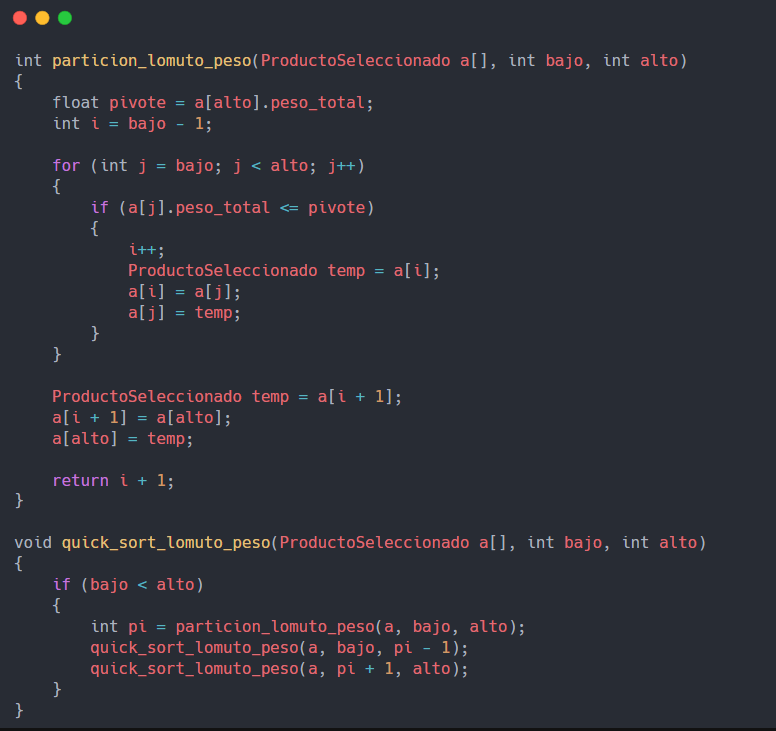
En el mejor caso (reparto uniforme, todos los buckets tienen pocos elementos):

Tmejor(n) = O(m + n)

En el caso promedio (reparto uniforme):

Tpromedio(n) = O(m + n log(n/m))

## Quick sort (lomuto) (Ordenación por peso - asignados)



**Descripción**

En nuestro proyecto utilicé una variante del algoritmo Quick Sort llamada partición de Lomuto para ordenar los productos seleccionados (ProductoSeleccionado) por su peso\_total.

La función quick\_sort\_lomuto\_peso() recibe un arreglo de productos, y los índices bajo y alto que definen el rango a ordenar. Dentro, utiliza una función auxiliar llamada particion\_lomuto\_peso() que selecciona como pivote el último elemento del arreglo y organiza los productos de forma que los más livianos queden a la izquierda y los más pesados a la derecha del pivote. Este proceso se repite recursivamente en cada subarreglo.

**Justificación del uso de Quick Sort (Lomuto)**

Decidimos usar Quick Sort con partición Lomuto porque es un algoritmo eficiente en la práctica para ordenar arreglos grandes. Aunque su peor caso es O(n^2) en la mayoría de los casos se comporta con una complejidad promedio de O(nlogn). Esta implementación es sencilla, clara y bien adaptada a mi estructura de datos.

Además, como estamos ordenando en memoria un arreglo de productos asignados (que ya fue filtrado previamente), no necesitamos estabilidad de ordenación ni manejo especial para claves repetidas. Quick Sort cumple con el rendimiento esperado en este contexto.

**Comparación con otras opciones**

|  |  |  |  |
| --- | --- | --- | --- |
| **Algoritmo** | **Complejidad promedio** | **Estable** | **Comentario** |
| **Quick Sort (Lomuto)** | O(nlog⁡n)O(n \log n) | ❌ | Rápido y simple de implementar |
| **Merge Sort** | O(nlog⁡n)O(n \log n) | ✅ | Estable y seguro, pero requiere más memoria |
| **Heap Sort** | O(nlog⁡n)O(n \log n) | ❌ | Buen rendimiento, pero más complejo |
| **Counting Sort** | O(n+k)O(n + k) | ✅ | Muy eficiente solo si el rango de valores es pequeño |

Para ordenar por peso, donde los valores pueden ser flotantes, algoritmos como Counting Sort no son viables sin una transformación adicional. Merge Sort sería una buena alternativa si necesitara una ordenación estable, pero Quick Sort es más rápido en promedio y suficiente para mis propósitos.

**Complejidad**

La recurrencia del algoritmo es:

Texto

El contenido generado por IA puede ser incorrecto.

y el peor caso es: *T(n − 1) + cn*

Expandimos varias veces:

Texto, Carta

El contenido generado por IA puede ser incorrecto.

Después de k=3 iteraciones llegamos a la conclusión que:

Imagen que contiene Texto

El contenido generado por IA puede ser incorrecto.

El proceso termina cuando n/2^k = 1 y k = log2 n:

*T(n) = nT(1) + cn log2 n*

*T(n) = Θ(n log n)*

Complejidad en el caso promedio y el mejor caso:

*Θ(n log n)*

En el peor caso:

Texto, Carta

El contenido generado por IA puede ser incorrecto.

Por lo tanto:

*T(n) = Θ(n^2)*

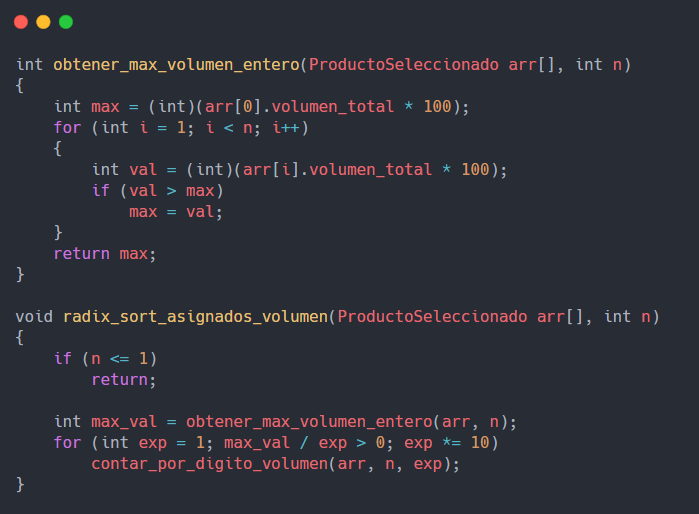
Entonces las complejidades finales son:

Mejor caso: *Θ(n log n)*

Caso promedio: *Θ(n log n)*

Peor caso: *Θ(n^2)*

## Radix sort (Ordenación por peso - asignados)



**Descripción**

En nuestro proyecto necesitábamos una manera eficiente de ordenar los productos asignados a los camiones según su volumen total. Para esto implementamos el algoritmo Radix Sort, el cual se adapta muy bien cuando se ordenan números enteros dentro de un rango limitado y cuando se requiere alta eficiencia.

Debido a que los volúmenes en nuestro sistema son valores flotantes (float), convertimos cada volumen a un número entero multiplicándolo por 100 (para conservar dos decimales) y así evitar problemas de precisión. Luego, el algoritmo ordena los productos dígito por dígito, empezando por la unidad y avanzando hacia las cifras más significativas (decenas, centenas, etc.).

Internamente, en cada pasada del algoritmo aplicamos Counting Sort, ya que es un algoritmo estable y eficiente para subordenar por cada dígito. Esto nos permite garantizar que los productos con el mismo volumen parcial conserven su orden relativo.

El procedimiento completo está implementado en la función radix\_sort\_asignados\_volumen, que recorre el arreglo de productos asignados (si\_seleccionados) y los organiza en orden creciente de volumen total.

**Justificación del uso de Radix Sort**

Elegimos Radix Sort por las siguientes razones:

1. Eficiencia en volúmenes numéricos pequeños: En nuestra aplicación, los productos tienen volúmenes dentro de un rango acotado (entre 1 y 50 m³), lo que hace que Radix Sort funcione en tiempo casi lineal (O(n·k)).
2. Estabilidad: Como Radix Sort es un algoritmo estable, respeta el orden relativo entre productos con volúmenes iguales. Esto es importante si más adelante combinamos criterios de ordenamiento (por ejemplo, volumen y nombre).
3. Evita comparaciones directas de flotantes: Convertimos el volumen a enteros y trabajamos con dígitos, lo que nos permite evitar errores de precisión y simplificar el algoritmo.
4. Escalabilidad: Si en el futuro aumentamos la cantidad de productos o localidades, Radix Sort nos ofrece un rendimiento mucho más predecible que otros algoritmos basados en comparación.
5. Integración natural con nuestro sistema: Al contar ya con estructuras que agrupan productos por asignación, fue sencillo aplicar Radix Sort únicamente sobre los productos ya asignados (si\_seleccionados), sin afectar otras operaciones del sistema.

**Comparación con otras opciones**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algoritmo** | **Estable** | **Complejidad** | **Uso en UVS Express** | **Justificación** |
| **Radix Sort** | ✅ | O(n·k) | Volumen (asignados) | Ideal para campos numéricos con rango limitado |
| **Counting Sort** | ✅ | O(n + k) | Volumen (no asignados) | Eficiente pero menos flexible para volúmenes más variados |
| **Merge Sort** | ✅ | O(n log n) | Peso (no asignados) | Muy útil para mantener orden estable por otro criterio |
| **Quick Sort** | ❌ | O(n log n) | Peso (asignados) | Rápido pero no conserva orden relativo (no estable) |
| **Heap Sort** | ❌ | O(n log n) | Nombre (no asignados) | Buena opción para strings, aunque con menor prioridad en volumen |

**Complejidad**

obtener max volumen entero recorre el arreglo una sola vez:

*Tmax(n) = O(n)*

contar por digito volumen también recorre el arreglo una cantidad constante de veces por dígito siendo básicamente un counting sort:

Texto, Carta

El contenido generado por IA puede ser incorrecto.

*c1 = t3, c2 = t2, c3 = t1, c4 = t0*

*T(n) = d · n · c1 + d · k · c2 + d · c3 + c4*

*T(n) = k1 · n + k2*

*Tcontar(n) = O(n+k) referente a la parte del counting sort*

*T(n) = O(d · (n + k))*

Dando como resultado en el mejor, peor y caso promedio:

*T(n) = O(d x n)*

## Búsqueda de producto por nombre (Búsqueda Binaria Recursiva)

**Texto

El contenido generado por IA puede ser incorrecto.**

**Descripción**

Para permitir al usuario localizar un producto específico dentro del sistema UVS Express, se implementó un algoritmo de búsqueda binaria, aplicado sobre un arreglo de estructuras Producto previamente ordenado alfabéticamente por nombre. Este proceso inicia solicitando al usuario el nombre exacto del producto, y luego ejecuta una búsqueda recursiva sobre el arreglo productosArreglo.

La búsqueda binaria divide el arreglo en mitades sucesivas. En cada paso compara el nombre proporcionado con el nombre del producto ubicado en la posición media del segmento actual. Si el nombre coincide, el producto es encontrado. Si no, el algoritmo determina si debe buscar en la mitad izquierda o derecha según el orden lexicográfico. Este enfoque garantiza una localización eficiente, especialmente cuando se manejan grandes volúmenes de productos.

Para garantizar que la búsqueda sea válida, se ejecuta previamente la función ordenar\_productos\_por\_nombre() utilizando qsort, asegurando que el arreglo esté ordenado alfabéticamente como lo requiere la búsqueda binaria.

**Justificación del algoritmo**

Se eligió la búsqueda binaria por las siguientes razones:

* Alta eficiencia temporal: Tiene una complejidad de O(log n), lo que significa que incluso con cientos o miles de productos, el número de comparaciones necesarias es muy bajo. Esto mejora el rendimiento frente a búsquedas lineales de complejidad O(n).
* Consistencia y precisión: La comparación se realiza usando strcmp, lo que permite comparar nombres de forma lexicográfica, asegurando una correspondencia exacta.
* Facilidad de implementación: A pesar de su eficiencia, la búsqueda binaria es sencilla de implementar y mantener en un sistema con estructura de datos simple como un arreglo.
* Escenario ideal de uso: Como los productos no cambian de nombre constantemente y se ordenan al inicio del programa, el entorno es perfecto para aplicar este algoritmo con gran efectividad.

**Comparación con otras alternativas**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Método de Búsqueda** | **Complejidad** | **Requiere orden** | **Ventajas** | **Desventajas** |
| **Búsqueda binaria** | O(log n) | ✅ Sí | Muy rápida en arreglos ordenados | Requiere orden previo |
| **Búsqueda lineal** | O(n) | ❌ No | Sencilla de implementar | Lenta en listas grandes |
| **Hashing (tabla hash)** | O(1) (prom.) | ❌ No | Extremadamente rápida | Mayor uso de memoria y complejidad extra |
| **Árbol binario (BST)** | O(log n)\* | ❌ No | Búsqueda eficiente con inserciones rápidas | Requiere estructuras dinámicas y balanceo |

**Complejidad**

La función de recurrencia queda como: T(n) = T(n/2) + c

Imagen que contiene Texto

El contenido generado por IA puede ser incorrecto.



2^k = n y k = log2 n

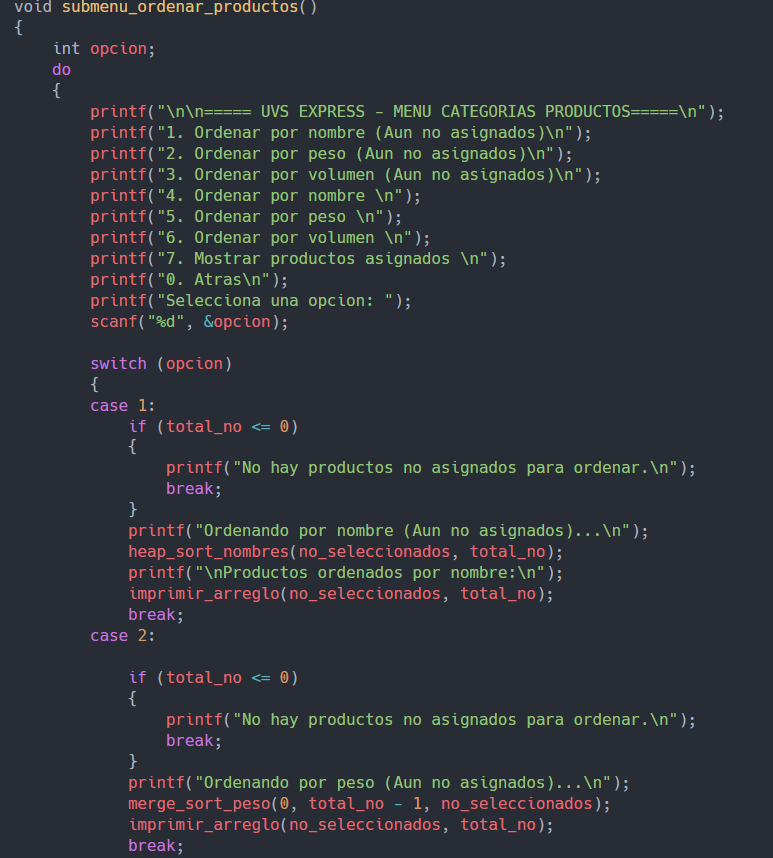
entonces tenemos *T(n) = T(1) + c · log2 n*

dando como resultado *T(n) ∈ O(log n)* en el peor caso y el caso promedio

la única diferencia seria que el primer elemento al entrar a la búsqueda binaria sea exactamente el elemento del medio por lo cual la complejidad será en el mejor caso de:

*T(n) ∈ O(1)*

## Submenu ordenar productos



**Descripción**

La función submenu\_ordenar\_productos() representa un submenú interactivo que ofrece al usuario diferentes formas de ordenar los productos que han sido asignados o no asignados a los camiones. Está diseñada como parte del menú principal del sistema logístico "UVS EXPRESS" y permite visualizar los productos según distintos criterios relevantes: nombre, peso o volumen.

El submenú funciona mediante un ciclo do-while que presenta continuamente las opciones al usuario hasta que este decida salir (opción 0). Para cada criterio y estado (asignado/no asignado), se utiliza un algoritmo de ordenación distinto:

* **Opción 1**: Ordenar productos no asignados por nombre → heap\_sort\_nombres()
* **Opción 2**: Ordenar productos no asignados por peso → mergeSort()
* **Opción 3**: Ordenar productos no asignados por volumen → counting\_sort\_volumen()
* **Opción 4**: Ordenar productos asignados por nombre → hash\_sort\_asignados\_nombre()
* **Opción 5**: Ordenar productos asignados por peso → quick\_sort\_lomuto\_peso()
* **Opción 6**: Ordenar productos asignados por volumen → radix\_sort\_asignados\_volumen()
* **Opción 7**: Mostrar productos asignados sin ordenar
* **Opción 0**: Salir del submenú

Antes de ejecutar cualquier ordenación, se valida que haya productos disponibles (asignados o no asignados), y se muestra un mensaje si no es así.

**Justificación**

La existencia de este submenú cumple varias funciones clave en el sistema:

1. Modularidad y usabilidad: El menú separado permite al usuario enfocarse exclusivamente en tareas relacionadas con el análisis y visualización de productos, sin interferir con otras funcionalidades como rutas o asignaciones.
2. Flexibilidad de análisis: Ofrecer múltiples criterios de ordenación ayuda a entender mejor la distribución y características de los productos transportados (por nombre para identificación, por peso para logística, por volumen para espacio).
3. Elección estratégica de algoritmos:
   * Se usaron algoritmos distintos según el tipo de dato y la eficiencia esperada:
     + Heap Sort para ordenación lexicográfica por nombre (bueno para inserciones dinámicas con propiedades de prioridad).
     + Merge Sort y Quick Sort para peso, que pueden manejar bien datos con alta variabilidad.
     + Counting Sort y Radix Sort para volumen, aprovechando que los volúmenes están transformados en enteros pequeños, ideal para ordenación por dígitos o conteo.
4. Interactividad en consola: Al usar estructuras como scanf y printf, el usuario tiene control directo sobre la ejecución, facilitando pruebas y validaciones durante la simulación o desarrollo.

## Menu Principal



**Descripción de la función menuPrincipal()**

La función menuPrincipal() actúa como punto de entrada principal para la interacción del usuario con el sistema UVS EXPRESS. Se trata de un menú en consola implementado mediante un bucle do-while que presenta repetidamente una lista de opciones hasta que el usuario decide salir del programa (opción 0).

Opciones disponibles:

1. Asignar productos óptimamente a camiones  
   → Llama a asignacion\_optima\_productos\_camiones() para distribuir productos en camiones según restricciones de peso y volumen usando programación dinámica.
2. Optimizar rutas por tiempo  
   → Llama a optimizar\_y\_asignar\_rutas(1) para calcular las rutas más rápidas usando el algoritmo de Dijkstra.
3. Optimizar rutas por distancia  
   → Llama a optimizar\_y\_asignar\_rutas(2) para encontrar rutas más cortas en kilómetros usando Floyd-Warshall.
4. Ordenar productos (Submenú)  
   → Llama a submenu\_ordenar\_productos() donde se permite ordenar productos asignados o no asignados por diferentes criterios.
5. Buscar producto por nombre  
   → Llama a buscar\_producto\_por\_nombre(), que realiza una búsqueda binaria sobre el arreglo de productos previamente ordenado.
6. Salir  
   → Finaliza el programa.

Cada opción se gestiona mediante una estructura switch-case, y en caso de ingresar una opción inválida, el sistema lo notifica.

**Justificación**

La inclusión de menuPrincipal() tiene justificación técnica y de experiencia de usuario:

1. Organización centralizada: Esta función organiza todas las funcionalidades clave del sistema en un solo punto, sirviendo como interfaz principal de navegación.
2. Modularidad: Cada opción activa una función específica y modular, lo que mejora el mantenimiento del código y facilita la lectura y escalabilidad del sistema.
3. Interacción simple y clara: Mediante el uso de scanf y printf, se permite una interacción directa desde la consola, ideal para simulaciones o prototipos de sistemas logísticos.
4. Facilita pruebas individuales: Al permitir ejecutar cada módulo por separado (asignación, rutas, ordenamientos, búsqueda), se pueden probar y validar funcionalidades de forma independiente durante el desarrollo.
5. Refuerza el flujo de lógica de negocio: El orden de las opciones refleja un flujo lógico:
   * Primero se asignan productos,
   * luego se optimizan rutas,
   * después se organizan los productos,
   * y finalmente se consulta la información individual.

# Conclusión

Este proyecto fue una experiencia muy completa donde pudimos aplicar muchos de los conocimientos que hemos visto a lo largo del curso, pero llevándolos a una situación realista. La idea de simular una empresa de logística que reparte productos en distintas localidades usando camiones con capacidades limitadas nos obligó a pensar en soluciones eficientes, bien organizadas y con sentido práctico.

Una de las cosas más importantes que aprendimos fue cómo asignar productos de manera óptima a los camiones. Para eso usamos un algoritmo basado en programación dinámica, que nos permitió tomar decisiones considerando tanto el peso como el volumen de los productos, algo que no se puede hacer con soluciones simples. Esto nos ayudó a entender cómo resolver problemas que parecen difíciles al principio, dividiéndolos en partes más pequeñas y evitando repetir cálculos innecesarios.

También trabajamos en la parte de rutas. Implementamos dos algoritmos de caminos mínimos: Dijkstra, que usamos para minimizar el tiempo de recorrido, y Floyd-Warshall, que usamos cuando queríamos minimizar la distancia. Con esto entendimos que no siempre la ruta más corta en kilómetros es la más rápida, y que las decisiones pueden cambiar dependiendo del criterio que se quiera optimizar.

Otro aprendizaje importante fue el uso de algoritmos de ordenamiento, ya que tuvimos que usar uno distinto para cada combinación de productos (asignados o no asignados) y criterio (nombre, peso o volumen). Esto nos ayudó a conocer mejor varios algoritmos como merge sort, quick sort, radix sort, counting sort y heap sort, y ver en qué situaciones funcionan mejor. Además, tuvimos que asegurarnos de que todos fueran lo suficientemente eficientes para cumplir con las restricciones del proyecto.

En la parte de búsqueda, implementamos búsqueda binaria, lo cual nos permitió encontrar productos por nombre de forma muy rápida. Pero también entendimos que para que este tipo de búsqueda funcione, es necesario tener los datos ordenados previamente.

Además del código, también aprendimos a organizarnos como equipo, a modular nuestro programa en funciones claras, y a revisar constantemente que todo estuviera funcionando bien. Nos dimos cuenta de que no basta con que algo funcione, también debe ser eficiente, claro y fácil de mantener.

En conclusión, este proyecto nos permitió aplicar todo lo que aprendimos en clase, pero también nos enseñó a pensar de manera más estructurada, a analizar cada decisión que tomamos en el código, y a buscar siempre la mejor solución posible para cada parte del problema. Sin duda fue un reto, pero también una experiencia muy valiosa y satisfactoria.