FACULTAD DE INGENIERÍA Y CIENCIAS

UNIVERSIDAD ADOLFO IBÁÑEZ

Informe de Diseño y Justificación de la Solución “Tarea 1”

Benjamín Fernández
Alonso Gil
Javier Quezada

Profesor: Maria Loreto Arriagada
Sección: 1
Fecha: 22/03/2025

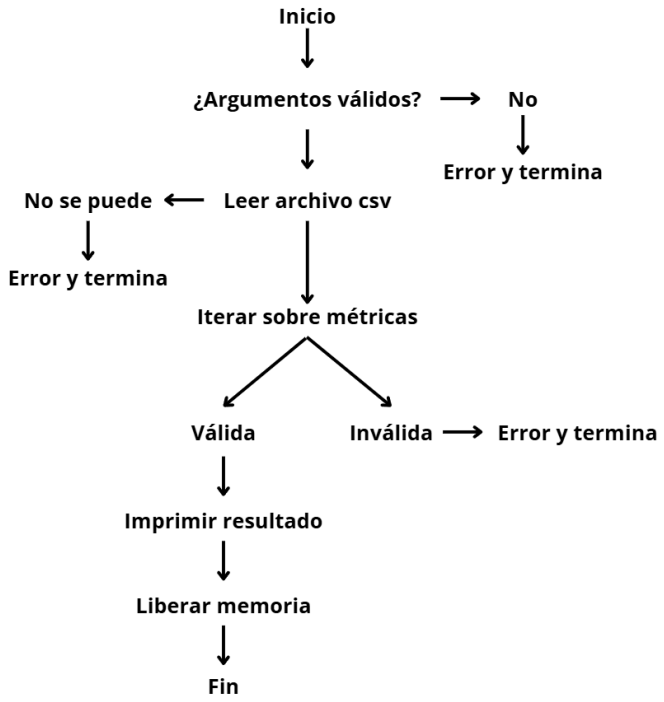
1. Objetivo

El objetivo del presente informe es describir la organización del código, las estructuras de datos empleadas y la justificación de la modularidad en la solución implementada para procesar un archivo CSV con información sobre pedidos de pizzas. Además, se explica la interacción entre los archivos del proyecto y se brindan detalles sobre las decisiones de diseño tomadas.

2. Diagrama de Flujo General

1. Inicio
2. Verifica argumentos de entrada
    - Si son insuficientes, muestra un mensaje de error y termina.

3. Lee el archivo CSV (read_csv)
    - Si hay un error en la lectura, muestra un mensaje y termina.
4. Itera sobre las métricas solicitadas
    - Llama a ***execute_metric*** para cada métrica.
    - Si la métrica es válida, imprime el resultado.
    - Si no es válida, muestra un mensaje de error.
5. Libera memoria y finaliza el programa.



3. Organización del Código

El código fue diseñado de manera modular para facilitar su mantenimiento y extensibilidad. A continuación, se muestran los diferentes archivos principales para la funcionalidad del programa

- **Makefile:** Es el responsable de compilar usando gcc todos los códigos de una vez, de esta manera no se tiene que compilar uno a uno los códigos.
- **metrics.c**: Contiene las funciones para calcular o medir las diferentes métricas que el usuario pida.
- **metrics.h**: Define las estructuras de datos y declaraciones de funciones necesarias para las métricas.
- **csv_reader.c** : Contiene las funciones para leer y parsear el archivo CSV.
- **csv_reader.h:** Contiene la estructura de como es el archivo CSV.
- **main.c**: Punto de entrada del programa, donde se manejan los datos de entrada y se llaman a las funciones..

Esta estructura modular permite separar la lógica de procesamiento de datos de las operaciones de entrada y salida, mejorando la claridad del código.

4. Estructuras de Datos Empleadas

Struct Order: Esta estructura se usa para representar cada pedido registrado en el archivo CSV. Contiene los siguientes campos:

```c
typedef struct {
    int pizza_id;
    int order_id;
    char pizza_name_id[MAX_STRING];
    int quantity;
    char order_date[MAX_STRING];
    char order_time[MAX_STRING];
    float unit_price;
    float total_price;
    char pizza_size[MAX_STRING];
    char pizza_category[MAX_STRING];
    char pizza_ingredients[MAX_STRING * 2];
    char pizza_name[MAX_STRING];
} Order;
```

Esta estructura permite almacenar todas las variables posibles del archivo CSV, facilitando el conteo y el análisis de las métricas pedidas.

Continuando, la estructura del código para las diferentes métricas fue:

PMS Y PLS:

1. **Pizza más vendida (PMS)** → Pizza con mayor cantidad de ventas junto al número total de unidades vendidas.
2. **Pizza menos vendida (PLS)** → Pizza con menor cantidad de ventas junto al número total de unidades vendidas.

Para calcular la pizza más vendida (PMS: Pizza Más Solicitada) y la pizza menos vendida (PLS: Pizza Menos Solicitada), se utilizó la estructura PizzaCount, que permite almacenar el nombre de cada pizza junto con la cantidad de unidades vendidas.

```c
// Estructura auxiliar para contar las pizzas mas y menos vendidas
typedef struct {
    char pizza_name[MAX_STRING];
    int quantity;
} PizzaCount;
```

Esta estructura facilita la organización de los datos y permite realizar el conteo de manera eficiente. Para evitar duplicados al contar las ventas, se implementó la función find_pizza, que recorre el arreglo de pizzas y compara los nombres con strcmp. Si encuentra una coincidencia, devuelve el índice; de lo contrario, retorna -1, indicando que es una nueva pizza.

```c
int find_pizza(PizzaCount *pizzas, int size, char *name) {
    for (int i = 0; i < size; i++) {
        if (strcmp(pizzas[i].pizza_name, name) == 0) {
            return i;
        }
    }
    return -1;
}
```

La función calculate_pms_pls recibe las órdenes y calcula la pizza más y menos vendida. Si no hay órdenes válidas, asigna el mensaje "No hay pizzas registradas" tal y como se ve en código.

```c
void calculate_pms_pls(int order_count, Order *orders, char **pms, char **pls) {
    if (order_count <= 0 || orders == NULL) {
        *pms = strdup("No hay pizzas registradas");
        *pls = strdup("No hay pizzas registradas");
        return;
    }
```

Luego, se reserva memoria con “calloc” para almacenar todas las pizzas únicas y sus cantidades.

```
PizzaCount *pizzas = calloc(order_count, sizeof(PizzaCount));
if (!pizzas) {
    fprintf(stderr, "Error: No se pudo asignar memoria.\n");
    exit(EXIT_FAILURE);
}
```

Se recorre la lista de órdenes para contar cuántas veces se ha vendido cada pizza. Si el nombre de la pizza está vacío, se ignora la orden.

```
int pizza_count = 0;

// Contar las ventas de cada tipo de pizza
for (int i = 0; i < order_count; i++) {
    if (strlen(orders[i].pizza_name) == 0) {
        printf("Orden ignorada: Nombre vacío.\n");
        continue;  // Ignora pedidos con nombre vacío
    }
                                    int pizza_count
    int index = find_pizza(pizzas, pizza_count, orders[i].pizza_name);
    if (index == -1) { // Nueva pizza
        strcpy(pizzas[pizza_count].pizza_name, orders[i].pizza_name);
        pizzas[pizza_count].quantity = orders[i].quantity; // Registra aunque sea 0
        pizza_count++;
    } else { // Pizza existente, sumamos la cantidad
        pizzas[index].quantity += orders[i].quantity;
    }
}
```

Si después del filtrado no hay pizzas registradas, se retorna el mensaje correspondiente y se libera la memoria.

```
if (pizza_count == 0) {
    *pms = strdup("No hay pizzas registradas");
    *pls = strdup("No hay pizzas registradas");
    free(pizzas);
    return;
}
```

Para encontrar la pizza más y menos vendida, se recorre el arreglo pizzas comparando las cantidades vendidas. Se actualizan los índices ***max_index*** y ***min_index*** según corresponda.

```
int max_index = 0, min_index = 0;
for (int i = 1; i < pizza_count; i++) {
    if (pizzas[i].quantity > pizzas[max_index].quantity) {
        max_index = i;
    }
    if (pizzas[i].quantity < pizzas[min_index].quantity) {
        min_index = i;
    }
}
```

Finalmente, se asignan los nombres de la pizza más y menos vendida a los punteros pms y pls, y se libera la memoria utilizada.

```
    // Asignar resultados
    *pms = strdup(pizzas[max_index].pizza_name);
    *pls = strdup(pizzas[min_index].pizza_name);

    free(pizzas);
}
```

DMS, DLS, DMSP, DLSP:

Para calcular las métricas basadas en la fecha, se utilizó la siguiente estructura y código:

1. **Fecha con más ventas en términos de dinero (DMS)** → Fecha con mayor recaudación junto al monto total.

2. **Fecha con menos ventas en términos de dinero (DLS)** → Fecha con menor recaudación junto al monto total.

3. **Fecha con más ventas en términos de cantidad de pizzas (DMSP)** → Fecha con mayor cantidad de pizzas vendidas junto al total de pizzas.

4. **Fecha con menos ventas en términos de cantidad de pizzas (DLSP)** → Fecha con menor cantidad de pizzas vendidas junto al total de pizzas.

Estructura DataSales

Esta estructura se emplea para almacenar información sobre las ventas en una fecha específica:

- **order_date**: Almacena la fecha de la venta (como una cadena de caracteres).
- **total_sales**: Acumula el total de dinero recaudado en esa fecha.
- **total_pizzas**: Cuenta la cantidad total de pizzas vendidas en esa fecha.

```c
// dms, dls, dmsp, dlsp: Métricas para analizar ventas por fecha
// Estructura auxiliar para contar ventas por fecha
typedef struct {
    char order_date[MAX_STRING];  // Fecha de la venta
    double total_sales;           // Total de dinero recaudado por las ventas en esa fecha
    int total_pizzas;             // Cantidad total de pizzas vendidas en esa fecha
} DateSales;
```

Función auxiliar para búsqueda de fecha

Se utiliza una función auxiliar que busca una fecha en un array de **DateSales** y devuelve su índice.

Funcionamiento:

- Recorre el array **sales** de tamaño **size**.
- Si encuentra una coincidencia con date, devuelve su índice.
- Si no encuentra la fecha, devuelve **-1**.

```c
// Función auxiliar para buscar una fecha en el array de ventas
// Recibe un array de ventas, el tamaño del array y la fecha a buscar
int find_date(DateSales *sales, int size, char *date) {
    // Recorre las ventas registradas para encontrar la fecha
    for (int i = 0; i < size; i++) {
        // Si la fecha ya existe, retorna el índice donde se encuentra
        if (strcmp(sales[i].order_date, date) == 0) {
            return i;
        }
    }
    return -1;  // Si no se encuentra la fecha, retorna -1
}
```

Manejo de errores iniciales

- Si no hay órdenes (**order_count <= 0**) o la lista de órdenes es **NULL**, la función asigna el mensaje de error "No hay ventas registradas" a cada métrica y finaliza la ejecución.

```c
// Función para calcular las métricas dms, dls, dmsp y dlsp
void calculate_date_metrics(int order_count, Order *orders, char **dms, char **dls, char **dmsp, char **dlsp) {
    // Verificar si no hay órdenes o si las órdenes son NULL
    if (order_count <= 0 || orders == NULL) {
        // Asignar valores de error si no hay ventas
        *dms = strdup("No hay ventas registradas");
        *dls = strdup("No hay ventas registradas");
        *dmsp = strdup("No hay ventas registradas");
        *dlsp = strdup("No hay ventas registradas");
        return;
    }
```

Reserva de memoria para almacenamiento de ventas por fecha

- Se reserva memoria para almacenar las ventas agrupadas por fecha.
- Si la asignación de memoria falla, se muestra un mensaje de error y el programa se detiene.

```c
// Asignar memoria para almacenar las ventas por fecha
DateSales *sales = calloc(order_count, sizeof(DateSales));
if (!sales) {
    // Si no se puede asignar memoria, mostrar error
    fprintf(stderr, "Error: No se pudo asignar memoria.\n");
    exit(EXIT_FAILURE);
}
```

Procesamiento de órdenes

- Se recorre la lista de órdenes y se agrupan las ventas por fecha.
- Si la fecha ya existe en **sales**, se suman las ventas y la cantidad de pizzas.
- Si la fecha no existe, se crea una nueva entrada en **sales**.

```c
int sales_count = 0;  // Contador para las ventas registradas

// Contar las ventas totales y la cantidad de pizzas por fecha
for (int i = 0; i < order_count; i++) {
    if (strlen(orders[i].order_date) == 0) {
        continue;  // Si la orden no tiene fecha, ignorarla
    }

    // Buscar si la fecha ya existe en el array de ventas
    int index = find_date(sales, sales_count, orders[i].order_date);
    if (index == -1) {  // Nueva fecha, se agrega al arreglo
        strcpy(sales[sales_count].order_date, orders[i].order_date);
        sales[sales_count].total_sales = orders[i].total_price;  // Acumular el dinero de la venta
        sales[sales_count].total_pizzas = orders[i].quantity;    // Acumular la cantidad de pizzas vendidas
        sales_count++;  // Incrementar el contador de fechas
    } else {  // Fecha existente, se suman las ventas y las pizzas
        sales[index].total_sales += orders[i].total_price;  // Sumar el total de dinero
        sales[index].total_pizzas += orders[i].quantity;    // Sumar las pizzas
    }
}
```

Manejo de error si no hubo ventas válidas

- Si **sales_count == 0**, significa que no se registraron ventas válidas.
- Se asignan mensajes de error a las métricas y se libera la memoria.

```c
// Si no hay ventas válidas después de procesar todas las órdenes
if (sales_count == 0) {
    // Asignar valores de error a todas las métricas
    *dms = strdup("No hay ventas registradas");
    *dls = strdup("No hay ventas registradas");
    *dmsp = strdup("No hay ventas registradas");
    *dlsp = strdup("No hay ventas registradas");
    free(sales);  // Liberar la memoria asignada
    return;
}
```

Identificación de fechas con más y menos ventas

Se recorren todas las fechas en **sales** para encontrar:

- **max_sales_index**: Fecha con más ventas en dinero.
- **min_sales_index**: Fecha con menos ventas en dinero.
- **max_pizzas_index**: Fecha con más pizzas vendidas.
- **min_pizzas_index**: Fecha con menos pizzas vendidas.

```
// Recorrer todas las fechas para encontrar las que tienen más y menos ventas
for (int i = 1; i < sales_count; i++) {
    // Comparar las ventas totales para encontrar el máximo y mínimo
    if (sales[i].total_sales > sales[max_sales_index].total_sales) {
        max_sales_index = i;  // Actualizar el índice de la fecha con más ventas
    }
    if (sales[i].total_sales < sales[min_sales_index].total_sales) {
        min_sales_index = i;  // Actualizar el índice de la fecha con menos ventas
    }

    // Comparar la cantidad de pizzas vendidas para encontrar el máximo y mínimo
    if (sales[i].total_pizzas > sales[max_pizzas_index].total_pizzas) {
        max_pizzas_index = i;  // Actualizar el índice de la fecha con más pizzas
    }
    if (sales[i].total_pizzas < sales[min_pizzas_index].total_pizzas) {
        min_pizzas_index = i;  // Actualizar el índice de la fecha con menos pizzas
    }
}
```

Asignación de resultados

- Se reserva memoria para cada métrica (**DMS, DLS, DMSP, DLSP**).
- Se almacena el resultado en formato:
    - "Fecha con más/menos ventas: **<fecha>**, Total: **<valor>**".

```
    // Asignar resultados para cada métrica
    *dms = malloc(MAX_STRING * sizeof(char));  // Reservar memoria para el resultado de la métrica dms
    snprintf(*dms, MAX_STRING, "Fecha con más ventas (en dinero): %s, Total: %.2f",
        sales[max_sales_index].order_date, sales[max_sales_index].total_sales);

    *dls = malloc(MAX_STRING * sizeof(char));  // Reservar (char [52])"Fecha con menos ventas (en dinero): %s, Total: %.2f"
    snprintf(*dls, MAX_STRING, "Fecha con menos ventas (en dinero): %s, Total: %.2f",
        sales[min_sales_index].order_date, sales[min_sales_index].total_sales);

    *dmsp = malloc(MAX_STRING * sizeof(char));  // Reservar memoria para el resultado de la métrica dmsp
    snprintf(*dmsp, MAX_STRING, "Fecha con más ventas (en cantidad de pizzas): %s, Total: %d",
        sales[max_pizzas_index].order_date, sales[max_pizzas_index].total_pizzas);

    *dlsp = malloc(MAX_STRING * sizeof(char));  // Reservar memoria para el resultado de la métrica dlsp
    snprintf(*dlsp, MAX_STRING, "Fecha con menos ventas (en cantidad de pizzas): %s, Total: %d",
        sales[min_pizzas_index].order_date, sales[min_pizzas_index].total_pizzas);

    free(sales);  // Liberar la memoria asignada para las ventas por fecha
}
```

Liberación de memoria

Por último, se libera la memoria con **free(sales)**, asegurando una gestión eficiente de los recursos.

APO, APD:

Para calcular las métricas basadas en el promedio, se utilizó la siguiente estructura y código:

1. **Promedio de pizzas por orden (apo)** → Número total de pizzas vendidas dividido entre el número de órdenes.

2. **Promedio de pizzas por día (apd)** → Número total de pizzas vendidas dividido entre el número de días distintos en los que hubo pedidos.

Verificación de datos de entrada

Antes de calcular las métricas, se valida si los datos son correctos:

- Si **order_count** (cantidad de órdenes) es menor o igual a **0** o si el puntero **orders** es **NULL**, se asigna **-1** a **apo** y **apd** como señal de error, y la función termina.

```c
// Nueva métrica: Promedio de pizzas por orden (apo) y Promedio de pizzas por día (apd)
void calculate_promedio(int order_count, Order *orders, double *apo, double *apd) {
    // Verificamos si el número de órdenes es válido o si las órdenes son NULL
    if (order_count <= 0 || orders == NULL) {
        *apo = -1;  // Si hay un error, asignamos -1 a apo
        *apd = -1;  // Si hay un error, asignamos -1 a apd
        return;     // Salimos de la función
    }
```

Cálculo del promedio de pizzas por orden (apo)

- Se inicializa **total_pizzas = 0.**

- Se recorre la lista de órdenes sumando la cantidad de pizzas vendidas (**orders[i].quantity**).

- Se calcula el promedio dividiendo **total_pizzas** entre **order_count**.

```c
// Calcular el promedio de pizzas por orden (apo)
int total_pizzas = 0;  // Variable para acumular el total de pizzas

// Recorremos todas las órdenes
for (int i = 0; i < order_count; i++) {
    total_pizzas += orders[i].quantity;  // Sumamos la cantidad de pizzas de cada orden
}

*apo = (double) total_pizzas / order_count;  // Calculamos el promedio de pizzas por orden
```

Cálculo del promedio de pizzas por día (apd)

Dado que cada orden tiene una fecha (**order_date**), el código agrupa las órdenes por días únicos para determinar el promedio de pizzas vendidas por día.

a) Uso de un arreglo **sales** para agrupar ventas por día.

- Se reserva memoria para **sales**, que almacenará la cantidad total de pizzas vendidas por cada día único.

- Se inicializa **sales_count = 0** (contador de días únicos).

```
// Creamos un arreglo para contar las ventas por fecha
DateSales *sales = calloc(order_count, sizeof(DateSales));  // Reservamos memoria para el arreglo de ventas
int sales_count = 0;  // Variable para contar el número de días únicos
```

b) Recorrer todas las órdenes y agruparlas por fecha

Para cada orden:

1. Se ignoran las órdenes con una fecha vacía ("").

2. Se verifica si la fecha ya está en **sales** usando la función **find_date()**.

    - Si **no está**, se agrega la fecha y se registra la cantidad de pizzas vendidas ese día.

    - Si **ya está**, simplemente se suman las pizzas a la fecha existente.

```
// Recorremos todas las órdenes para contar las ventas por fecha
for (int i = 0; i < order_count; i++) {
    // Si la fecha de la orden está vacía, la ignoramos
    if (strlen(orders[i].order_date) == 0) continue;

    // Buscamos si la fecha ya está registrada en el arreglo sales
    int index = find_date(sales, sales_count, orders[i].order_date);

    // Si la fecha no está registrada, la agregamos al arreglo sales
    if (index == -1) {
        strcpy(sales[sales_count].order_date, orders[i].order_date);  // Guardamos la fecha
        sales[sales_count].total_pizzas = orders[i].quantity;  // Guardamos la cantidad de pizzas para esa fecha
        sales_count++;  // Aumentamos el contador de fechas
    } else {
        // Si la fecha ya está registrada, acumulamos las pizzas vendidas en esa fecha
        sales[index].total_pizzas += orders[i].quantity;
    }
}
```

c) Calcular el promedio de pizzas por día

1. **total_days = sales_count**, que representa el número de días únicos con pedidos.

2. Se recorre **sales** sumando todas las pizzas vendidas (**total_pizzas_day**).

3. Se calcula **apd = total_pizzas_day / total_days**.

```
    // Ahora calculamos el total de días y el total de pizzas vendidas en esos días
    int total_days = sales_count;  // El número total de días es igual a las fechas únicas encontradas
    int total_pizzas_day = 0;  // Variable para acumular el total de pizzas por día

    // Sumamos las pizzas vendidas en cada día
    for (int i = 0; i < sales_count; i++) {
        total_pizzas_day += sales[i].total_pizzas;  // Acumulamos el total de pizzas vendidas por día
    }

    // Calculamos el promedio de pizzas vendidas por día
    *apd = (double) total_pizzas_day / total_days;

    // Liberamos la memoria utilizada por el arreglo sales
    free(sales);
}
```

Liberación de memoria

Por último, se libera la memoria con **free(sales)**, asegurando una gestión eficiente de los recursos.

IMS

Ingrediente más vendido (ims) → Ingrediente de las pizzas que más fue vendido en el periodo.

Primero, struct se usa para almacenar el nombre de un ingrediente y la cantidad total de veces que ha sido vendido en las órdenes.

```
typedef struct {
    char ingredient[MAX_STRING];
    int count;
} IngredientCount;
```

Siguiendo en el código, esta función elimina los espacios en blanco al inicio y al final de una cadena. Se usa para asegurarse de que los nombres de los ingredientes se comparen correctamente sin espacios extra. Es decir, si por ejemplo los datos de entrada están escritos como “ Queso “ la salida los deja como “Queso”.

```c
void trim_spaces(char *str) {
    int start = 0;
    int end = strlen(str) - 1;

    // Eliminar los espacios al principio
    while (str[start] == ' ' || str[start] == '\t') {
        start++;
    }

    // Eliminar los espacios al final
    while (end >= start && (str[end] == ' ' || str[end] == '\t')) {
        end--;
    }

    // Mover la cadena recortada al inicio
    for (int i = start; i <= end; i++) {
        str[i - start] = str[i];
    }
    str[end - start + 1] = '\0';  // Terminar la cadena
}
```

Luego, está la función más importante. Su objetivo es encontrar el ingrediente más vendido analizando todas las órdenes.

```c
void calculate_ims(int order_count, Order *orders, char **ims_result) {
```

Luego este código hace que cada ingrediente se recorra y se busque en la lista ingredientes. Si ya existe, se aumenta su contador. Si no existe, se añade a la lista.

```c
while (token) {
    // Eliminar espacios extra al principio y al final de cada ingrediente
    trim_spaces(token);

    int found = 0;
    // Buscar si el ingrediente ya está en la lista
    for (int j = 0; j < ing_count; j++) {
        if (strcmp(ingredients[j].ingredient, token) == 0) {
            ingredients[j].count += orders[i].quantity;
            found = 1;
            break;
        }
    }
    // Si el ingrediente no se encontró, agregarlo a la lista
    if (!found) {
        strcpy(ingredients[ing_count].ingredient, token);
        ingredients[ing_count].count = orders[i].quantity;
        ing_count++;
    }

    // Obtener el siguiente ingrediente
    token = strtok(NULL, ",");
}
```

Por último, aquí se recorre la lista “ingredients” para encontrar el que tenga la mayor cantidad de ventas y luego se asigna memoria para la variable ims_result y se almacena en ella el nombre del ingrediente más vendido.

```
// Encontrar el ingrediente más vendido
int max_count = 0;
char most_sold[MAX_STRING];
for (int i = 0; i < ing_count; i++) {
    if (ingredients[i].count > max_count) {
        max_count = ingredients[i].count;
        strcpy(most_sold, ingredients[i].ingredient);
    }
}

// Asignar el resultado al puntero de salida
*ims_result = malloc(MAX_STRING);
snprintf(*ims_result, MAX_STRING, "Ingrediente más vendido: %s", most_sold);
```

HP

vendidas por categoría (HP) → Cantidad de pizzas por categoría vendidas.

Aquí se define una estructura llamada ***CategoryCount*** que tiene dos campos:

- ***category:*** Un arreglo de caracteres para almacenar el nombre de la categoría de la pizza (por ejemplo, "Margarita", "Pepperoni").

- ***count:*** Un entero que mantiene el conteo de la cantidad de pizzas vendidas en esa categoría.

```
// Función para calcular cantidad de pizzas por categoría
void calculate_hp(int order_count, Order *orders, char **hp_result) {
    typedef struct {
        char category[MAX_STRING];
        int count;
    } CategoryCount;
```

- ***categories***: Es un arreglo de 20 elementos de tipo ***CategoryCount.*** Este arreglo almacenará las categorías de pizzas y su respectivo conteo.

- ***cat_count:*** Es un contador que lleva la cuenta de cuántas categorías únicas de pizzas se han encontrado hasta el momento. Inicialmente está en 0.

```
CategoryCount categories[20];
int cat_count = 0;
```

El segundo bucle for recorre las categorías existentes (***cat_count*** es el número de categorías ya registradas).

Si la categoría de la pizza en la orden actual coincide con una categoría previamente almacenada (***strcmp*** compara cadenas), entonces:

- Se actualiza el contador de esa categoría sumando la cantidad de pizzas vendidas en la orden actual (***categories[j].count += orders[i].quantity***).
- Se marca ***found = 1*** para indicar que la categoría ya fue encontrada y procesada, y se rompe el bucle.
- Se copia el nombre de la categoría de la pizza actual en el campo ***category*** del nuevo elemento del arreglo ***categories.***
- Se asigna la cantidad de pizzas de esa categoría (***orders[i].quantity***) al campo ***count.***
- Se incrementa el contador ***cat_count*** para que en la próxima iteración se pueda añadir una nueva categoría en la siguiente posición disponible.

```
for (int i = 0; i < order_count; i++) {
    int found = 0;
    for (int j = 0; j < cat_count; j++) {
        if (strcmp(categories[j].category, orders[i].pizza_category) == 0) {
            categories[j].count += orders[i].quantity;
            found = 1;
            break;
        }
    }
    if (!found) {
        strcpy(categories[cat_count].category, orders[i].pizza_category);
        categories[cat_count].count = orders[i].quantity;
        cat_count++;
    }
}
```

Aquí se asigna memoria para la cadena ***hp_result***, que es donde se almacenará el resultado final con las categorías y sus cantidades. Se reserva ***MAX_STRING*** bytes para asegurarse de que haya suficiente espacio.

Se inicializa la cadena ***hp_result*** con un encabezado que indica el propósito del resultado: "Categorías de pizzas vendidas:". Esto servirá como encabezado en el resultado final.

```
    *hp_result = malloc(MAX_STRING);
    strcpy(*hp_result, "Categorías de pizzas vendidas:\n");
    for (int i = 0; i < cat_count; i++) {
        char buffer[50];
        snprintf(buffer, sizeof(buffer), "%s: %d\n", categories[i].category, categories[i].count);
        strcat(*hp_result, buffer);
    }
}
```

Finalmente, utilizamos esta función que recibe tres parámetros:

- const char *metric: Un puntero a una cadena de caracteres que especifica el nombre de la métrica que se desea calcular y devolver.
- int order_count: Un número entero que indica la cantidad total de órdenes disponibles.
- Order *orders: Un puntero a un arreglo de estructuras Order, el cual contiene la información de cada una de las órdenes.

El objetivo de la función es mostrar los resultados de las métricas solicitadas según los parámetros proporcionados.

```
// Función para ejecutar métricas solicitadas.
char* execute_metric(const char *metric, int order_count, Order *orders) {
    static char *pms_result = NULL;
    static char *pls_result = NULL;
    static char *dms_result = NULL;
    static char *dls_result = NULL;
    static char *dmsp_result = NULL;
    static char *dlsp_result = NULL;
    static double apo_result = -1;
    static double apd_result = -1;
    static char *ims_result = NULL;
    static char *hp_result = NULL;


    // Solo calcular si aún no se ha hecho
    if (!pms_result || !pls_result) {
        calculate_pms_pls(order_count, orders, &pms_result, &pls_result);
    }
    if (!dms_result || !dls_result || !dmsp_result || !dlsp_result) {
        calculate_date_metrics(order_count, orders, &dms_result, &dls_result, &dmsp_result, &dlsp_re
    }
    if (apo_result == -1 || apd_result == -1) {
        calculate_promedio(order_count, orders, &apo_result, &apd_result);
    }
    if (!ims_result) {
        calculate_ims(order_count, orders, &ims_result);
    }
```

```c
char *result = malloc(MAX_STRING * sizeof(char));
if (!result) {
    fprintf(stderr, "Error: No se pudo asignar memoria.\n");
    exit(EXIT_FAILURE);
}

if (strcmp(metric, "pms") == 0) {
    snprintf(result, MAX_STRING, "Pizza más vendida: %s", pms_result);
} else if (strcmp(metric, "pls") == 0) {
    snprintf(result, MAX_STRING, "Pizza menos vendida: %s", pls_result);
} else if (strcmp(metric, "dms") == 0) {
    snprintf(result, MAX_STRING, "Fecha con más ventas: %s", dms_result);
} else if (strcmp(metric, "dls") == 0) {
    snprintf(result, MAX_STRING, "%s", dls_result);
} else if (strcmp(metric, "dmsp") == 0) {
    snprintf(result, MAX_STRING, "%s", dmsp_result);
} else if (strcmp(metric, "dlsp") == 0) {
    snprintf(result, MAX_STRING, "%s", dlsp_result);
} else if (strcmp(metric, "apo") == 0) {
    snprintf(result, MAX_STRING, "Promedio de pizzas por orden: %.2f", apo_result);
} else if (strcmp(metric, "apd") == 0) {
    snprintf(result, MAX_STRING, "Promedio de pizzas por día: %.2f", apd_result);
} else if (strcmp(metric, "ims") == 0) {
    snprintf(result, MAX_STRING, "%s", ims_result);
} else if (strcmp(metric, "hp") == 0) {
    snprintf(result, MAX_STRING, "%s", hp_result);
} else {
    snprintf(result, MAX_STRING, "Métrica no implementada: %s", metric);
```

```c
    } else if (strcmp(metric, "apo") == 0) {
        snprintf(result, MAX_STRING, "Promedio de pizzas por orden: %.2f", apo_result);
    } else if (strcmp(metric, "apd") == 0) {
        snprintf(result, MAX_STRING, "Promedio de pizzas por día: %.2f", apd_result);
    } else if (strcmp(metric, "ims") == 0) {
        snprintf(result, MAX_STRING, "%s", ims_result);
    } else if (strcmp(metric, "hp") == 0) {
        snprintf(result, MAX_STRING, "%s", hp_result);
    } else {
        snprintf(result, MAX_STRING, "Métrica no implementada: %s", metric);
    }
    return result;
}
```

Para la lectura del csv también utilizamos punteros:

- const char *filename: Este parámetro es un puntero a una cadena de caracteres (char*), y se utiliza para pasar el nombre del archivo CSV que se va a abrir y leer. Como mencionamos antes, en C, las cadenas de texto son manejadas como arreglos de caracteres, y los punteros son la forma de pasarlas entre funciones.
- Order **orders: Este parámetro es un puntero a puntero a una estructura Order. El propósito de este puntero doble es permitir que la función read_csv asigne memoria dinámicamente para un arreglo de estructuras Order y pueda modificar el puntero original en la función principal. Al usar *orders = malloc(count * sizeof(Order)), la función cambia el valor del puntero orders para que apunte a un arreglo de estructuras Order recién creado. Esto significa que el puntero a orders será modificado fuera de la función, permitiendo que la función almacene la información de las órdenes en ese arreglo.

```c
int read_csv(const char *filename, Order **orders) {
    FILE *file = fopen(filename, "r");
    if (!file) {
        perror("Error opening file");
        return -1;
    }

    char line[MAX_LINE_LENGTH];
    int count = 0;

    // Contar líneas (excluyendo la cabecera)
    while (fgets(line, sizeof(line), file)) count++;
    rewind(file);

    // Asignar memoria para las órdenes
    *orders = malloc(count * sizeof(Order));
    if (!*orders) {
        fclose(file);
        perror("Memory allocation failed");
        return -1;
    }
```

5. Razones de Diseño

El diseño del código se fundamenta en varias decisiones clave que garantizan su eficiencia, modularidad y facilidad de mantenimiento.

En primer lugar, se implementó una estructura modular dividiendo el código en múltiples archivos. Esta organización permite asignar a cada módulo una responsabilidad específica, lo que facilita su prueba y mantenimiento. Gracias a esta separación, se mejora la legibilidad del código y se optimiza el proceso de depuración.

En cuanto al procesamiento del archivo CSV, se decidió leer línea por línea y descomponer los datos en tokens. Este método permite identificar correctamente cada campo, asegurando la integridad de la información y evitando posibles errores en la lectura del archivo.

El almacenamiento de ingredientes se resolvió mediante cadenas de caracteres dentro de la estructura **Order**. Dado que no es necesario realizar operaciones complejas sobre los ingredientes, esta elección simplifica la gestión de memoria y evita el uso de estructuras más complejas que podrían añadir sobrecarga innecesaria al sistema.

Finalmente, para manejar los nombres de las pizzas de manera dinámica y evitar limitaciones de memoria, se emplearon punteros a cadenas de caracteres (**char**

pointers). Esta estrategia permite un uso eficiente de los recursos, ya que la memoria se asigna dinámicamente según sea necesario, evitando desperdicio de espacio y proporcionando mayor flexibilidad en la manipulación de los datos.

6. Interacción entre Archivos

El proyecto está compuesto por varios archivos principales, cada uno con una función específica que contribuye al correcto funcionamiento del programa.

Primeramente, el módulo encargado de la lectura y procesamiento del archivo CSV está compuesto por los archivos **csv_reader.c** y **csv_reader.h**. En este módulo se implementa la funcionalidad para leer y parsear los datos de entrada. La función principal toma el archivo CSV proporcionado, extrae la información relevante y la almacena en una estructura de datos adecuada para su posterior procesamiento.

Por otro lado, el cálculo de métricas se encuentra en el módulo conformado por **metrics.c** y **metrics.h**. Este módulo implementa las funciones necesarias para entregar las respuestas a las métricas que pida el usuario.

El archivo **main.c** representa el punto de entrada del programa. Su principal responsabilidad es llamar a todas las funciones (**csv_reader.c**, **csv_reader.h, metrics.c** y **metrics.h**) y finalmente, mostrar los resultados obtenidos. Este archivo actúa como coordinador del flujo de ejecución, asegurando la correcta interacción entre los diferentes módulos del sistema.

La comunicación entre los módulos sigue una estructura bien definida. El archivo **main.c** obtiene los datos de los pedidos a través del módulo **csv_reader.c** y posteriormente los procesa mediante las funciones definidas en **metrics.c**. Esta separación clara de responsabilidades facilita el mantenimiento del código y permite realizar futuras modificaciones con mayor flexibilidad y menor riesgo de afectar otras partes del sistema.

7. Sección de reflexiones finales o autoevaluación

Alonso Gil: Lo más complejo fue aprender el lenguaje y utilizarlo, ya que nunca lo había usado antes y presenta complejidades distintas a las de otros lenguajes con

los que estaba familiarizado. Además, el hecho de que no se ejecute de forma nativa en Windows complicó aún más el proceso, ya que instalé erróneamente herramientas que no eran necesarias. En cuanto a la complejidad de la tarea, esta se centró principalmente en la utilización del lenguaje y su aplicación al contexto específico.

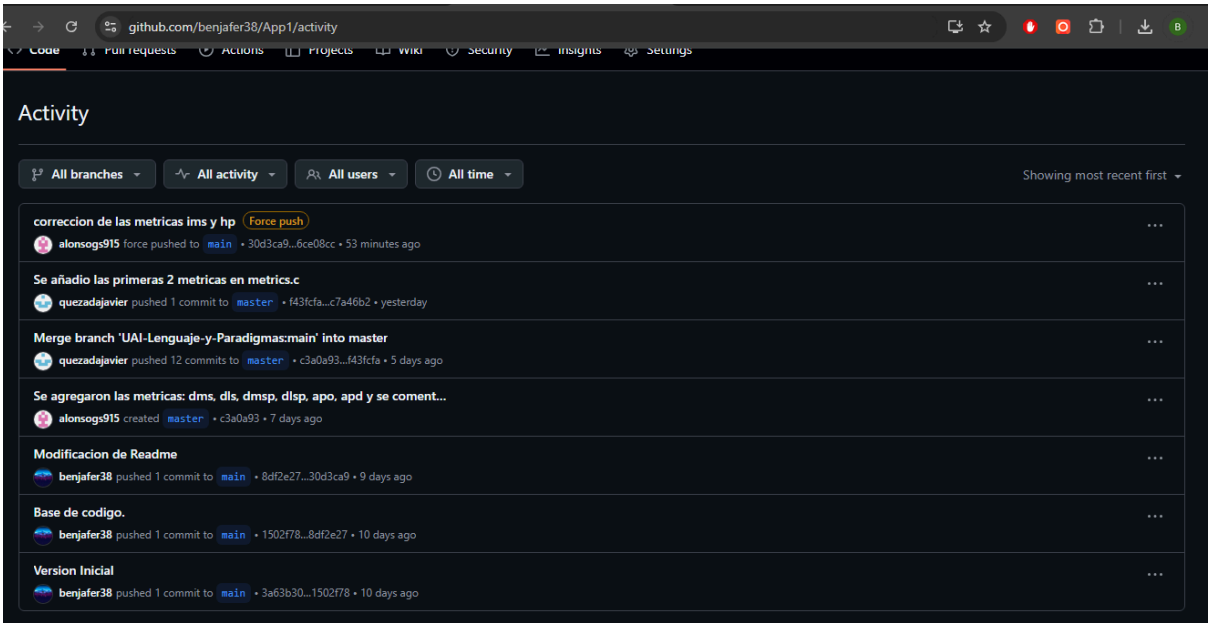
Enfrentamos los errores realizando pruebas constantes, ejecutando el código en conjunto y apoyándonos en herramientas de inteligencia artificial para identificar y corregir fallos.
Aprendimos sobre el manejo adecuado de variables, el procesamiento eficiente de datos y la importancia de liberar correctamente la memoria para garantizar un código óptimo y con buenas prácticas.También, una complicación es comprender el código de los demás compañeros y seguir tanto el flujo del código como la lógica utilizada detrás.

Benjamín Fernández: Al comenzar este proyecto, enfrentamos un gran desafío ya que ninguno de los integrantes tenía experiencia previa programando en C. La estructura inicial del código nos tomó muchas horas de análisis y pruebas, ya que organizar correctamente los módulos y entender cómo trabajar con archivos CSV en C fue una tarea compleja. Sin embargo, una vez que logramos comprender el funcionamiento de las métricas, el desarrollo del código se volvió más fluido y menos demandante.

Uno de los mayores problemas que enfrentamos fue el manejo de los ingredientes dentro del archivo CSV. Como los ingredientes están separados por comas, el lenguaje C los reconocía como nuevos campos, lo que generaba errores en la lectura de los datos. Para solucionar este problema, tuvimos que modificar la forma en que el programa procesaba cada línea del archivo, asegurándonos de que los ingredientes se interpretaran correctamente sin afectar el resto de los datos.

De este trabajo me llevo una reflexión importante, la evolución de los lenguajes de programación a lo largo del tiempo es muy grande. Reflexionando, nos dimos cuenta de que realizar esta tarea en Python habría sido mucho más sencillo, ya que en C, es necesario configurar un compilador, descargar herramientas adicionales para ejecutar el código en Windows y además gestionar múltiples archivos con código. En cambio, en Python, basta con escribir un código e incluir el archivo CSV, lo que demuestra la gran evolución que hubo en este campo.

Adicionalmente, por alguna razón, mis commits se borraron de la tarea, pero quiero aclarar que sí los realicé y fueron la base sobre la cual se comenzo a desarrollar el código. Adjunto una captura de pantalla como evidencia de que los commits existieron y fueron fundamentales para el desarrollo de la tarea.



Javier Quezada: De las dificultades que pudimos relacionar, es que tuvimos un inicio tardío en poder programar por un lenguaje nuevo de programación que tiene ciertos componentes muy distintos a lo que es python, que viene a ser un lenguaje mucho más simple frente a lo que es C, y como poder compilarlo bien en el computador para poder programar, y esto también tuvo sus dificultades, ya que cada vez que se debía correr el codigo para verificar si esté llevaba errores había que iniciarlo desde el terminal, seguido de un código específico, lo que hizo más lento la experiencia de realizar esta tarea de Pizzeria.

Ahora bien, ya centrando en la parte del código como tal, como equipo pudimos adaptarnos más a la programación en C, viendo los parámetros y la secuencia que debe seguir el código, aunque generando errores de vez en cuando por los puntos y coma que hay que colocar, donde muchas veces se olvidaban y el terminal no dejaba compilar, además de todo esto pudimos denotar la gran dificultad de escribir dentro de este lenguaje de programación en lugar de python o Sql, ya que el primero tiene un lenguaje de programación mucho más simple, y el segundo es un lenguaje específico para el manejo de datos, aprendimos que C, es la base de la programación que hay hoy en día, para desenvolver aún más los programas que se requieran realizar para un proyecto, tarea, etc. debido a la forma más específica de seguir un depurado en el código.

8. Explicación de cómo usaron IA

Durante el desarrollo del proyecto, utilizamos Chat GPT principalmente para la solución y corrección de errores, lo que simplificó significativamente tanto el proceso de depuración como la escritura del código. Dado que el lenguaje era completamente nuevo para nosotros y presentaba una estructura más compleja en comparación con otros con los que estábamos familiarizados, contar con esta herramienta nos permitió identificar rápidamente los fallos y comprender qué estaba ocurriendo en cada caso. Además, la IA nos proporcionó explicaciones detalladas sobre el funcionamiento de distintos elementos del lenguaje, lo que facilitó nuestro aprendizaje y optimizó el desarrollo del proyecto. Gracias a esto, pudimos abordar con mayor eficiencia las dificultades propias del lenguaje C, como la gestión de memoria, el manejo adecuado de archivos CSV y la necesidad de realizar pruebas constantes para evitar errores en la compilación.