APÉNDICE 3
PROBLEMAS

1.1. Compañía de aviación


Análisis y solución

Una compañía de aviación requiere implementar un programa que, basado en un sistema de acumulación de millas, promueva la fidelización de sus clientes.

Cada vez que un cliente vuela a través de la compañía, recibe una cantidad de millas acumulables que, posteriormente, podrá canjear por vuelos sin costo a diferentes destinos. Cuanto mayor sea la cantidad de millas acumuladas, más importante serán los destinos o premios por los cuales las podrá canjear.

La compañía cuenta con las siguientes archivos: `CIUDADES.dat`, que contiene la descripción de las ciudades a las que vuela, `VUELOS.dat`, con la información de los vuelos que conectan las diferentes ciudades, y `RESERVAS.dat`, con las reservas que realizaron los clientes para volar en los diferentes vuelos.

La estructura de cada uno de estos archivos la vemos a continuación.

CIUDADES

```
struct Ciudad
{
   int idCiu;
   char descr[20];
   int millas;
};
```

VUELOS

```
struct Vuelo
{
   int idVue;
   int cap;
   int idOri; // idCiu origen
   int idDes; // idCiu origen
};
```

RESERVAS

```
struct Reserva
{
   int idCli;
   int idVue;
   int cant;
};
```

La operatoria es la siguiente: un pasajero que vuela de una ciudad a otra acumula una cantidad de millas equivalente a la diferencia entre las millas establecidas para cada una de esas ciudades, multiplicado por la cantidad de plazas reservadas. Esto únicamente si su reserva es aceptada.

Sólo se aceptarán las reservas de aquellos pasajeros que requieran una cantidad de plazas menor o igual a la disponibilidad actual del vuelo en cuestión. De lo contrario la reserva completa será rechazada.

Se pide informar:

1. Para cada ciudad, cantidad de grupos (familias) que la eligieron de destino.
2. Por cada vuelo, cantidad de plazas rechazadas, indicando también si el vuelo saldrá completo o incompleto.
3. Para cada cliente, total de millas acumuladas.

1.2. Torneo de fútbol

Análisis y solución

Se requiere un programa que informe sobre la posición actual de cada uno de los equipos que participan de un torneo de fútbol.

Contamos con los siguientes archivos: `RESULTADOS.dat`, con los resultados de los partidos que se jugaron durante la última fecha, y `EQUIPOS.dat` que contiene la información de los e-quipos que participan del torneo.

La estructura de cada uno de estos archivos la vemos a continuación.

RESULTADOS

```
struct Resultado
{
   int idEq1;
   int idEq2;
   int codRes;
   char estadio[20];
};
```

EQUIPOS

```
struct Equipo
{
   int idEq;
   char nombre[20];
   int puntos;
};
```

El valor del campo `codRes` (código de resultado) indica qué equipo ganó el partido. Si `codRes<0` significa que ganó el equipo identificado con `idEq1`. Si `codRes>0` el ganador fue el equipo identificado con `idEq2`. Finalmente, si `codRes` es 0 (cero) el partido resultó en empate.

El equipo ganador acumula 3 puntos. Si empataron le corresponde 1 punto cada uno. El perdedor no recibe puntos.

Se pide:

1. Informar la tabla de posiciones actualizada al día del proceso.
2. Informar, para cada estadio, cuántos partidos se jugaron y cuántos de estos partidos resultaron empatados.
3. Actualizar las puntuaciones en el archivo `EQUIPOS.dat`.

1.3. Emisión de tickets


Análisis y solución

Un comercio vende productos clasificados en diferentes rubros. Algunos rubros pueden estar en promoción, razón por la cual sus productos se ofrecerán al público por debajo de su valor habitual.

Se dispone de los siguientes archivos: `PRODUCTOS.dat`, con la información de cada uno de los productos comercializados, y `RUBROS.dat`, que describe los rubros y sus promociones.

PRODUCTOS

```
struct Producto
{
   int idProd;
   char descr[20];
   double precio;
   int idRub;
};
```

RUBROS

```
struct Rubro
{
   int idRub;
   char descr[20];
   double promo;
};
```

Se pide:

1. Por cada venta, emitir un `ticket`, con el formato que se detalla a continuación, agrupando los productos y sumando sus cantidades, ordenando los ítems alfabéticamente según la descripción de los productos.

Número de ticket: 99999

Producto	*Precio*	*c/Dto.*	*Cant.*	*Total*
xxxxxxxxxxxxxx	999,99	999,99	999	99999,99
xxxxxxxxxxxxxx	999,99	999,99	999	99999,99
xxxxxxxxxxxxxx	999,99	999,99	999	99999,99
			TOTAL:	99999,99

Ahorro por rubro:

Rubro	*Total*
xxxxxxxxxxxx	999,99
xxxxxxxxxxxx	999,99
xxxxxxxxxxxx	999,99
TOTAL:	999,99

2. Informar cuáles fueron los 10 productos más demandados, ordenando el listado decrecientemente según la cantidad demandada.

1.4. Inscripción en la facultad

Análisis y solución

Para agilizar su sistema de inscripción, una facultad requiere desarrollar un programa que procese los siguientes archivos: `INSCRIPCIONES.dat`, con las inscripciones de los estudiantes a los diferentes cursos, y `CURSOS.dat`, con la oferta de cursos disponibles donde se dictarán las diferentes materias.

El archivo INSCRIPCIONES se encuentra ordenado decrecientemente según la fecha de la inscripción.

INSCRIPCIONES

```
struct Inscripcion
{
   int idAlu;
   int idCur;
   Fecha fecha; // TAD
};
```

CURSOS

```
struct Curso
{
   int idCur;
   char turno; // M, T o N
   int cap;
   char materia[20];
};
```

Se pide informar:

1. Por cada alumno, el listado de materias en que su inscripción resultó rechazada por falta de capacidad en el curso.
2. Por cada materia, el listado de cursos donde, luego de procesar las inscripciones, quedaron cupos disponibles.
3. Generar el archivo `REASIGNACION.dat`, cuya estructura se describe más abajo, reasignando (siempre que sea posible) las inscripciones rechazadas a aquellos cursos que quedaron con cupos disponibles. Ordenado por `idAlu`.

```
struct Reasignacion
{
   int idAlu;
   int idCurReasig;
};
```

4. Generar el archivo `REVISION.dat`, con la estructura que se describe a continuación y la información de los alumnos aún tienen cursos pendientes de ser reasignados, ordenado ascendentemente por `idAlu`.

```
struct Revision
{
   int idAlu;
   char materia[20];
};
```

1.5. Streaming de música

Una compañía que brinda servicios de música por *streaming* requiere estadísticas que le permitan conocer las preferencias de sus abonados. Para esto disponemos de los siguientes archivos: `REPRODUCCIONES.dat`, con el historial de los álbumes que los abonados escucharon. `ARTISTAS.dat`, con la información de los diferentes artistas, y `ALBUMES.dat` que describe el catálogo de discos que ofrece la compañía.

Análisis y solución

REPRODUCCIONES

```
struct Reproduccion
{
   int idUsuario;
   int idAlbum; // ordenado
   int fecha;
   int minutos;
};
```

ALBUMES

```
struct Album
{
   int idAlbum; // ordenado
   int idArtista;
   char titulo[50];
   int duracion; // minutos
};
```

ARTISTA

```
struct Artista
{
   int idArtista; // ordenado
   char nombre[50];
};
```

Dada la naturaleza del contexto, todos los archivos tienen una cantidad de registros tal que hace imposible mantenerlos en memoria.

Se pide:

1. Un listado ordenado por álbum, indicando cuántas reproducciones completas tuvo. Cuántas estuvieron entre el 75% y el 100%, cuántas entre el 50% y el 75%, cuántas entre el 25% y el 50%, y cuántas reproducciones duraron menos del 25% del total del álbum.
2. Los 10 artistas cuyos álbumes se reprodujeron entre el 75% y el 100%.

1.6. Prestadores médicos

Análisis y solución

Se desea medir la evolución del rendimiento que tuvo un centro de salud, contrastando las prestaciones médicas que se realizaron durante los dos últimos años. Para esto, disponemos de los archivos: `PRESTA19.dat` y `PRESTA20.dat`, que contienen la información de las prácticas que realizaron los médicos y técnicos que trabajan (o trabajaron) en dicho centro de salud durante los años 2019 y 2020 respectivamente. Y el archivo `PRACTICAS.dat`, que describe el catálogo de todas prestaciones disponibles.

PRESTA19 / PRESTA20

```
struct Presta
{
   int idPres;
   int idPrac;
   FechaHora fechaHora; // TAD
   int minutos;
};
```

PRACTICA

```
struct Practica
{
   int idPrac;
   char descr[50];
};
```

Los archivos PRESTA19 y PRESTA20 se encentran ordenados ascendentemente por `idPres+fechaHora`. El tipo `FechaHora` es un TAD cuya API de funciones tenemos a nuestra disposición.

Se pide:

1. Un listado, ordenado decrecientemente por `idPres`, de los prestadores que se incorporaron en 2020.
2. Para los prestadores que sí trabajaron durante 2019 y 2020, un listado de las prácticas que realizaron; ordenado decrecientemente por `fechaHora`.
3. Para los prestadores que sólo trabajaron durante 2019, un listado indicando la cantidad de prácticas realizadas.

1.7. Imputación horas/proyecto

Análisis y solución

Una consultora que gestiona diversos proyectos requiere información acerca de cómo, sus empleados, distribuyen el tiempo de trabajo. La información se encuentra disponibles en los archivos: `PROYECTOS.dat`, que describe cada uno de los proyectos que gestiona la consultora. `EMPLEADOS.dat`, con los datos de los empleados, y `HORAS.dat`, con el detalle de las horas de trabajo que cada empleado indica haber trabajado en cada proyecto.

PROYECTOS

```
struct Proyecto
{
   int idProyecto;
   char descripcion[100];
   Fecha fechaInicio;
   int horasAsignadas;
   int horasImputadas;
};
```

HORAS

```
struct Hora
{
   int idEmpleado;
   int idProyecto;
   Fecha fecha;
   int horas;
   char tareas[200];
};
```

EMPLEADOS

```
struct Empleado
{
   int idEmpleado;
   char nombre[50];
};
```

Se pide:

1. Emitir un listado indicando, para cada proyecto, qué empleados trabajaron; y por cada uno, qué tareas desarrolló. Ordenado por proyecto, luego por empleado, y finalmente por fecha descendente.

```
Proyecto: xxxxxxxxxxxxx
        Empleado: xxxxxxxxxxxxxxxxxxxxx
        Fecha           Tarea desarrollada                  Horas
        99/99/9999      xxxxxxxxxxxxxxxxxxxxxxxxx           9999
        99/99/9999      xxxxxxxxxxxxxxxxxxxxxxxxx           9999
        99/99/9999      xxxxxxxxxxxxxxxxxxxxxxxxx           9999
        :               :                   :
```

2. Actualizar el archivo de proyectos e indicar cuáles, luego de procesar las novedades, quedaron excedidos en cantidad de horas.

1.8. Asistencia mecánica

Análisis y solución

Una empresa que brinda asistencia mecánica a sus abonados solicita desarrollar un programa que ayude con la gestión de sus servicios.

La operatoria es la siguiente: cuando un abonado requiere asistencia se comunica con la empresa. Indica su número de abonado (`idAbo`) y en qué zona está. Existen 10 zonas, numera-das de 0 a 9.

El operador que recibe la llamada debe verificar que el abonado tenga sus cuotas al día. Luego, lo colocará en una cola de espera y le informará qué tiempo aproximado deberá esperar.

Cuando un móvil finaliza una asistencia se comunica con la empresa e informa cuál es el número de caso que acaba de resolver.

El móvil libre pasará a una cola y quedará en espera hasta que un nuevo caso le sea asignado. El operador le informará, estimativamente, qué tiempo deberá esperar.

Luego de cada evento (llamada de un abonado o de un móvil) el sistema debe verificar si es posible realizar una asignación móvil/abonado. En tal caso, deberá notificar a ambos involucrados (abonado y móvil) vía SMS al celular.

El abonado debe recibir un SMS con el nombre del conductor del móvil que está en camino. El móvil debe recibir el nombre del abonado que lo está esperando.

Las siguientes funciones son parte de la biblioteca y están disponibles para su uso.

```
// envia un SMS al celular especificado como parametro
void notificarAsignacion(string celularDestino
                         ,int nroCaso
                         ,string nombre);

// retorna la hora actual expresada en milisegundos
int getTime();
```

MOVILES

```
struct Movil
{
   int idMovil;
   char conductor[100];
   int zona;
   char celular[50];
};
```

ABONADOS

```
struct Abonado
{
   int idAbo;
   char nombre[100];
   char celular[50];
   bool cuotasAlDia;
};
```

El número de caso inicial se ingresará por teclado. A partir de allí, a cada caso se le asignará un valor correlativo.

Se pide:

1. Desarrollar un programa interactivo que asista al operador durante toda la operatoria descripta más arriba.
   El programa estará esperando a que ocurra un evento. Si llama un abonado, el operador ingresará el valor 1. Si llama un móvil, ingresará 2. Para finalizar ingresará 3.
2. Por cada móvil, indicar los casos cuyo tiempo de atención estuvo por debajo del promedio de la zona.

1.9. Línea de cajas

Análisis y solución

Se requiere un programa para optimizar la atención en la línea de cajas de un supermercado. Para esto, ponen a nuestra disposición el archivo MOVIMIENTOS (cuya estructura veremos más abajo), en el cual hay un registro por cada persona que entra en la cola de una caja, y otro por cada persona que sale.

Los ingresos se representan con el carácter 'E' en el campo `mov`. Los egresos tienen una 'S' en dicho campo. El campo `hora` indica a qué hora se produjo el ingreso o egreso de la cola de la caja `caja`.

```
struct Mov
{
   int caja;
   char mov;  // 'E' => Entra, 'S' => Sale
   Hora hora; // TAD
};
```

Se pide, procesando el archivo de movimientos, informar:

1. Tiempo promedio de espera por caja.
2. Sumatoria del tiempo ocioso por caja.
3. Longitud máxima a la llegó la cola de cada caja.

1.10. Optimización de colas

Análisis y solución

Se requiere desarrollar un programa para ayudar a encontrar una relación óptima entre la cantidad de cajas que se abren en un supermercado, el tiempo que esperan los clientes en ser atendidos, y el tiempo que permanecen ociosos los empleados que atienden dichas cajas.

Para esto, disponemos del archivo MOVIMIENTOS, cuya estructura vemos a continuación, que describe qué cliente ingresó o egreso de una caja; y a qué hora se produjo dicho movimiento.

```
struct Mov
{
   int idCli;
   char mov;  // 'E' o 'S'
   Hora hora; // TAD
};
```

Se realizará una simulación con 3, 4, 7, 8 y 10 cajas abiertas. Cuando un cliente llega a la línea de cajas se colocará en la cola más corta. Si hubiese una o varias cajas sin cola, se ubicará en cualquiera.