

# APÉNDICE 3

## PROBLEMAS

### 1.1. Problemas

Los problemas que se describen a continuación deben resolverse dos veces. La primera vez será al finalizar el capítulo 4 (indicados con 4), utilizando (si fuera necesario) el TAD `Coll`. La segunda vez será al finalizar el capítulo 5 (indicados con 5), reemplazando (según corresponda) el TAD `Coll` por los TAD `Array`, `Map`, `List`, `Stack` y `Queue`.

#### 1.1.1. Compañía de aviación (4 y 5)

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.



*Análisis y solución*

La compañía cuenta con los 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 será así siempre y cuando su reserva sea 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.

### 1.1.2. Torneo de fútbol (4 y 5)

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 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 equipos que participan del torneo.



*Análisis y solución*

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`.

### 1.1.3. Emisión de tickets (4 y 5)

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.

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



Análisis y solución

#### PRODUCTOS

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

#### RUBROS

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

Por cada cliente se ingresará su `idCli`, y varios pares `{idProd, cant}`. Un `idProd=0` indicará el final de la compra del cliente. Un `idCli=0` indicará la finalización de la operatoria del comercio.

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
				<b>TOTAL: 99999,99</b>

Ahorro por rubro:

Rubro	Total
xxxxxxxxxxxxxx	999,99
xxxxxxxxxxxxxx	999,99
<b>TOTAL: 999,99</b>	

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

#### 1.1.4. Inscripción en la facultad (4 y 5)

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 ascendientemente según la fecha de la inscripción.



Análisis y solución

##### INSCRIPCIONES

```
struct Inscripcion
{
    int idAlu;
    int idCur;
    int fecha; // aaaammdd
};
```

##### 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];
};
```

### 1.1.5. Streaming de audiocuentos (4 y 5)

Un emprendimiento universitario registra y publica como *streaming* el audio proveniente de la lectura de cuentos, conformando una plataforma de *audiocuentos*.

Se dispone de los siguientes archivos:

#### REPRODUCCIONES

```
struct Reproduccion
{
    int idUsuario;
    int idCuento;
    int fecha;
    int minutos;
};
```

#### CUENTOS

```
struct Cuento
{
    int idCuento;
    int idRelator;
    int idAutor;
    char titulo[50];
    int duracion;
};
```

#### RELATORES

```
struct Relator
{
    int idRelator;
    char nombre[50];
};
```

Dada la naturaleza del contexto (emprendimiento universitario), todos los archivos tienen una cantidad acotada de registros.

Se pide:

1. Un listado ordenado por cuento, 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 cuento.
2. Los 10 relatores cuyas lecturas tuvieron la mayor cantidad de reproducciones entre el 75% y 100%, ordenado de mayor a menor por dicha cantidad.

### 1.1.6. Streaming de audiolibros (4 y 5)

Un emprendimiento universitario registra y publica como *streaming* el audio proveniente de la lectura de libros, conformando una plataforma de *audiolibros*.

Se dispone de los siguientes archivos:

#### REPRODUCCIONES

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

#### CUENTOS

```
struct Libro
{
    int idLibro;
    int idRelator;
    int idAutor;
    char titulo[50];
    int duracion;
};
```

#### RELATORES

```
struct Relator
{
    int idRelator;
    char nombre[50];
};
```

Dada la naturaleza del contexto (emprendimiento universitario), todos los archivos tienen una cantidad acotada de registros.

La duración de un libro puede llegar a ser de varias horas. Por esta razón, un muy probable que un mismo usuario realice varias reproducciones del mismo libro. En tal

caso, todas las reproducciones serán consecutivas, cada una comenzará desde donde se dejó de reproducir la anterior.

Se pide:

1. Un listado ordenado por libro, 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 cuento.
2. Los 10 relatores cuyas lecturas tuvieron la mayor cantidad de reproducciones entre el 75% y 100%, ordenado de mayor a menor por dicha cantidad.

### 1.1.7. Streaming de música (4 y 5)

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 tuvieron la mayor cantidad de reproducciones entre el 75% y 100%, ordenado de mayor a menor por dicha cantidad.

### 1.1.8. Prestadores médicos (4 y 5)

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.



Análisis y solución

PRESTA19 / PRESTA20

```
struct Presta
{
    int idPres;
    int idPrac;

    // aaaammddhmm
    long long fechaHora;
    int minutos;
};
```

PRACTICA

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

Los archivos `PRESTA19` y `PRESTA20` se encuentran ordenados ascendentemente por `idPres+fechaHora`.

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.

### 1.1.9. Imputación horas/proyecto (4 y 5)

Una consultora que gestiona diversos proyectos requiere información sobre cómo, los empleados, distribuyen el tiempo de trabajo. Los datos se encuentran disponibles en los siguientes archivos: `PROYECTOS.dat`, que describe los proyectos gestionados por la consultora, `EMPLEADOS.dat`, con los datos de los empleados, y `HORAS.dat`, con el detalle de las horas de trabajo que cada empleado dice haber trabajado en cada proyecto.



Análisis y solución

El archivo `HORAS` se encuentra ordenado ascendentemente por `fecha`. Es decir: los registros antiguos primero, y los más actuales después.

#### PROYECTOS

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

#### HORAS

```
struct Hora
{
    int idEmpleado;
    int idProyecto;
    int 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: xxxxxxxxxxxx
Empleado: xxxxxxxxxxxxxxxxxxxx
Fecha      Tarea desarrollada      Horas
99/99/9999 xxxxxxxxxxxxxxxxxxxxxxxx 9999
99/99/9999 xxxxxxxxxxxxxxxxxxxxxxxx 9999
99/99/9999 xxxxxxxxxxxxxxxxxxxxxxxx 9999
:           :                   :

```

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

### 1.1.10. Asistencia mecánica (5)

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

Cuando un abonado requiere asistencia se comunica con la empresa. Indica su número de abonado (`idAbo`) y en qué zona está. Existen 10 zonas, numeradas 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 el tiempo aproximado que 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 mensaje de texto (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.



Análisis y solución

Las siguientes funciones de biblioteca 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();
```

Las estructuras de los archivos MOVILES.dat y ABONADOS.dat son:

#### 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 el programa ingresará 3.
2. Al finalizar el programa indicar, por cada móvil, los casos cuyo tiempo de atención estuvo por debajo del promedio de la zona.

### 1.1.11. Línea de cajas (5)

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.dat` (cuya estructura ya veremos), 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 en la cola de la caja `caja`.



Análisis y solución

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

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.

### 1.1.12. Optimización de colas (5)

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.

Contamos con el archivo `MOVIMIENTOS.dat`, cuya estructura vemos a continuación, que describe qué cliente ingresó o egreso de una caja; y a qué hora se produjo dicho movimiento.



Análisis y solución

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

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.

La salida que espera obtener es:

<i>Cantidad de cajas:</i>	3	4	7	8	10
<i>Tiempo de espera:</i>	999	999	999	999	999
<i>Tiempo ocioso:</i>	999	999	999	999	999

### 1.1.13. Gastos por consorcio (4 y 5)

Un estudio que administra diversos consorcios requiere un programa que lo ayude para liquidar expensas.

En todos los casos, los departamentos de los consorcios se identifican con dígitos numéricos (1, 2, 3, etcétera), y todos los pisos de un consorcio tienen la misma cantidad de departamentos, distribuidos exactamente del mismo modo.



*Análisis y solución*

Disponemos de los archivos `GASTOS.dat` y `CONSORCIOS.dat`.

#### GASTOS

```
struct Gasto
{
    int idCons; // ver situacion (*)
    int fecha; // aaaammdd
    double importe;
    char categoria[20];
};
```

#### CONSORCIOS

```
struct Consorcio
{
    int idCons;
    char direccion[50];
    int cantPisos;
    int dtosPorPiso;
};
```

La función `mtsPorcentuales` que, dado un consorcio y un departamento, retorna la proporción que dicho departamento ocupa dentro del piso.

```
double mtsPorcentuales(int idCons,int depto);
```

Por ejemplo: si un tipo de departamento de un determinado consorcio ocupa el 28% del piso, la función retornará 0,28.

Se pide emitir un listado para cada consorcio, con el detalle que se indica más abajo, ordenado decrecientemente por la cantidad de metros porcentuales que cada departamento ocupa dentro del piso.

<i>Consorcio Calle: XXXXXXXXXXXXXXXXXXXX</i>		<i>Gastos totales: \$9999</i>
<i>Depto.</i>	<i>Mts.2.</i>	<i>Total a pagar (\$)</i>
3	33%	999999
1	30%	999999
2	25%	999999
:	:	:

(\*)

*Situación (A):* Considere que `GASTOS.dat` está ordenado por `idCons`.

*Situación (B):* Considere que `GASTOS.dat` no está ordenado.

#### 1.1.14. Gastos por consorcio (versión 2) (4 y 5)

Ídem anterior, pero el listado solicitado también debe mostrar cómo se distribuyen los gastos en función de las diferentes categorías. Debe aparecer ordenado decrecientemente por gasto, y decrecientemente por total a pagar.

*Consorcio Calle: XXXXXXXXXXXXXXXXXXXX*

<i>Categoría</i>	<i>Gastos</i>
Limpieza	9999
Electricidad	9999
Sueldos	9999
<b>TOTAL</b>	<b>\$99999</b>

Depto.	Mts.2.	Total a pagar (\$)
3	33%	999999
1	30%	999999
2	25%	999999
:	:	:

### 1.1.15. Gastos por consorcio (versión 3) (4 y 5)

Ídem anterior, pero no se dispone de la función `mtsPorcentuales`. En su reemplazo, se agrega el archivo `DISTRIBUCION.dat`, sin orden y con la siguiente estructura.

```
struct Distribucion
{
    int idCons;
    int tipoDto;
    double mtsPorc;
};
```

### 1.1.16. Obras de teatro (4 y 5)

Un sitio de Internet vende localidades para las obras de teatro que se encuentran en cartel en las diferentes salas de la ciudad.

Cada sala tiene varios sectores (platea, pulman, preferencial, etcétera), y cada sector admite una determinada capacidad de espectadores. Los sectores de las salas teatrales se numeran desde 1 y son correlativos.



Análisis y solución

Disponemos de `OBRAS.dat` y `TEATROS.dat`, con las siguientes estructuras.

OBRAS

```
struct Obra
{
    int idObra;
    char titulo[100];
    int fEstreno; // aaaammdd
    int idTeatro;
};
```

TEATROS

```
struct Teatro
{
    int idTeatro;
    char direccion[50];
    int capacidad;
    int sectores;
};
```



Contamos también con la función `funcionesProgramadas`, que retorna una colección de funciones según cuál sea el `idObra` que reciba como parámetro. Si provenimos de la primera parte del curso usaremos la primera versión de la función. En cambio, si ya desarrollamos el TAD `List`, usaremos la segunda versión.

```
// si provenimos de la primera parte del curso
Coll<Funcion> funcionesProgramadas(int idObra);

// si provenimos de la segunda parte del curso
List<Funcion> funcionesProgramadas(int idObra);
```

La estructura `Funcion` es la siguiente:

```
struct Funcion
{
    int idFuncion;
    int diaSem;      // 1=>Lunes, 2=>Martes, ...
    int hora;        // hhmm
};
```

NOTA: `idFuncion` es un valor único e irrepetible, independientemente de cuál sea la función y la obra de teatro.

Por cada reserva de localidades que un cliente genere a través del sitio Web, el sistema produce registro `Reserva`, cuya estructura es:

```
struct Reserva
{
    int idCliente;
    int idObra;
    int idFuncion;
    int sector; // sector de la sala; ej: Pullman, Platea...
    int cant;
};
```

Las siguientes funciones permiten interactuar con el sitio Web.

```
Reserva leerReserva();
bool continuarOperando();
```

La función `leerReserva` es *bloqueante*. Esto significa que se quedará esperando hasta que algún cliente haya ingresado los datos necesarios que permitan generar una nueva reserva. Por su parte, `continuarOperando` indica si se debe continuar aceptando (leyendo) nuevas reservas o no.

Finalmente, se dispone de la función `capacidadSector` que retorna la capacidad de un sector de una sala teatral.

```
int capacidadSector(int idTeatro, int sector);
```

Se pide: emitir un listado indicando la cantidad de reservas que quedaron excluidas por falta de capacidad (sólo se aceptarán reservas completas).

```
Obra (Titulo): XXXXXXXXXXXXXXXXXXXX
  Dia/Hora      Cantidad
  99/99:99      999
  99/99:99      999
  :              :
  :
```

### 1.1.17. Obras de teatro (versión 2) (4 y 5)

Ídem, pero ya no disponemos de ninguna de las funciones que proveía el enunciado anterior. En su reemplazo, aparecen los siguientes archivos: `RESERVAS.dat` (con la estructura de registro `Reserva` antes mencionada), `FUNCIONES.dat` y `SECTORES.dat`, cuyos registros tienen las estructuras que vemos a continuación:

#### FUNCIONES

```
struct Funcion
{
    int idFuncion;
```

#### SECTORES

```
struct Sector
{
    int idTeatro;
```

```
int idObra
int diaSem; // 1=>lun, 2=>mar,...
int hora;   // hhmm
};
```

```
int sector;
int cap;
};
```

### 1.1.18. Obras de teatro (versión 3) (5)

Un sitio de Internet permite reservar localidades para las diferentes obras que se exhiben en los teatros de la ciudad. Dichas reservas podrán ser aceptadas o rechazada por cuestiones de disponibilidad para el sector/función/obra.



Análisis y solución

*Operatoria y acotaciones:*

- Las reservas serán aceptadas siempre y cuando exista disponibilidad suficiente para cubrir la totalidad de ubicaciones requeridas por el cliente. De lo contrario se rechazarán.
- Las obras (*idObra*) no serán más de 1000, numeradas entre 1 y 1000.
- Sólo se programa una función por día de la semana, pudiendo no haber funciones todos los días.
- Ningún teatro tiene más de 10 sectores, numerados a partir de 1.

*Recursos, estructuras y funciones disponibles:*

```
struct Obra
{
    int idObra;
    char titu[100];
    int fEstreno; // aaaammdd
    int idTeatro;
};

struct Funcion
{
    int idFuncion;
    int diaSemana;
    int hora; // hhmm
};
```

```
struct Teatro
{
    int idTeatro;
    char nom[50];
    int capacidad;
    int sectores;
};

struct Reserva
{
    int idCli;
    int idObra;
    int dia;
    int sector;
    int cant;
};
```

`Obra getObra(int idObra);` Retorna los datos de una obra.

`Teatro getTeatro(int idTeatro);` Retorna los datos de un teatro.

`Reserva leerReserva();` Lee una reserva ingresada por el sitio.

`bool continuarOperando();` Determina si finaliza la operatoria del sitio.

`List<Funcion> getFuncionesObra(int idObra);` Retorna una lista enlazada donde cada nodo contiene con los datos de una `Funcion`.

`int capacidadSector(int idTeatro, int sector);` Retorna la capacidad del sector `sector` del teatro `idTeatro`. Dicho valor lo retornará sólo una vez. Las posteriores invocaciones retornarán un valor negativo.

`void notificarReserva(int idCli, bool acept);` Notifica al cliente acerca del rechazo o aceptación de su reserva de localidades.

Se pide:

1. Notificar a los clientes sobre la aceptación o el rechazo de sus reservas.
2. Imprimir un listado que indique la cantidad de localidades rechazadas por cada obra, sin discriminar por función ni sector.

<i>Obra</i>	<i>Rechazos</i>
6	999
15	999
18	999
23	999
:	:

### 1.1.19. Hospedaje en casas de familia (4 y 5)

Un sitio de Internet que ofrece alojamiento en casas de familia requiere aumentar las funcionalidades de sus servicios.

En el archivo `CARACTERISTICAS.dat`, que se describe más abajo, el valor del campo `idCaract` es una potencia de 2. Por ejemplo: 1=>Cercanía a la costa, 2=>Aire acondicionado, 4=>Parrilla, 8=>Pileta de natación, 16=>Cochera, etcétera. Cada potencia implica una característica diferente.



Análisis y solución

En el archivo `CASAS.dat`, si el valor del campo `idUsr`, es 0 (cero) significa que la casa en cuestión está disponible. De lo contrario indicará el *id* del usuario que la arrendó.

#### CARACTERISTICAS

```
struct Caract
{
    int idCaract;
    char descr[100];
};
```

#### CASAS

```
struct Casa
{
    int idCasa;
    char direcc[50];
    int idDueno;
    int caractMask;
    int idUsr;
};
```

Los usuarios buscan casas que tengan ciertas características. Cada búsqueda genera un registro del siguiente tipo:

```
struct Busqueda
{
    int idUsr;
    int caractMask;
    int dias;
    double tolerancia;
};
```

El campo `caractMask` es una máscara de bits de 4 bytes, donde cada bit indica la existencia o no de una determinada característica.

Se dispone de siguiente función:

```
double concordancia(int idCasa,int mask);
```

Esta función compara las características de una casa con las características que desea el potencial inquilino, y retorna un porcentaje de coincidencia; siendo 1 el 100%, 0,8 el 80%, etcétera.

Se dispone también de las funciones:

Busqueda leerBusqueda() y bool continuarOperando() a través de las cuales el programa puede interactuar con la operatoria del portal de Internet.

Se pide, por cada búsqueda, emitir un listado (ordenado decrecientemente por el porcentaje de concordancia) de todas las casas disponibles cuya concordancia está por encima del valor `tolerancia` del registro de la búsqueda. Por cada casa, se debe invocar a la función `void mostrarCasa(int idCasa)` que mostrará en la página Web las fotos, detalles y demás datos de interés para el usuario.

### 1.1.20. Canal de televisión (4 y 5)

Un canal de televisión requiere validar una propuesta de programación diaria. La misma se encuentra detallada en el archivo `PLANIFICACION.dat`, que sólo es un borrador de lo que será la programación definitiva. Su estructura, así como la del archivo `PROGRAMAS.dat` se describe a continuación.



Análisis y solución

#### PLANIFICACION

```
struct Planificacion
{
    int idPlanif;
    int idProg;
    int horaInicio;
    int minutoInicio;
};
```

#### PROGRAMAS

```
struct Programa
{
    int idProg;
    char titulo[100];
    int duracion; // minutos
    int atp; // 0=>no, 1=>si
};
```

En la programación propuesta podrían existir errores de dos tipos: superposición horaria, y programas que se emiten en un horario inadecuado, según sean ATP (Apto para Todo Público) o no.

En caso de existir superposición horaria, el programa que se deberá descartar será el posterior. Por ejemplo: Si el programa *p1* comienza a las 13 horas y dura 1 hora y media, y el programa *p2* comienza a las 14 horas, éste será descartado de la planificación propuesta.

Podría suceder que un programa se extienda hacia el día siguiente. Por ejemplo, si comienza a las 23:30 horas y dura 120 minutos. Finalizará a la 1:30 horas del día siguiente, situación que también podría ocasionar superposición horaria.

El horario ATP está establecido por ley, y rige entre las 7 y las 22 horas. No debería suceder que un programa para adultos se emita total o parcialmente dentro de dicho horario.

Recursos: se dispone de las siguientes funciones:

TAD `Hora` - Utilice (pero no programe) las funciones que considere necesarias.

`haySuperposicion` - retorna `true` si dos programas se superponen entre sí. Si `hi2 > hi1` será porque corresponde a un programa que comienza el siguiente día.

Parámetros:

- `Hora hi1` - Hora de inicio de un programa.
- `int dur` - Duración (en minutos) del programa que comienza a las `hi1`.
- `Hora hi2` - Hora de inicio de otro programa.

```
bool haySuperposicion(Hora hi1,int dur,Hora hi2);
```

`invadeHorarioATP` - que retorna `true` si un programa, o parte del mismo, invade el horario Apto para Todo Público.

Parámetros:

- `Hora hi` - Hora de inicio de un programa.
- `int dur` - Duración (en minutos) del programa que comienza a las `hi1`.

```
bool invadeHorarioATP(Hora h,int dur);
```

`calcularBache` - retorna la hora de inicio de un bache en la programación, y le asigna a `durb` la duración de dicho bache, en minutos.

Parámetros:

- Hora `hi1` – Hora de inicio de un programa.
- `int durp` – Duración (en minutos) del programa que comienza a las `hi1`.
- Hora `hi2` – Hora de inicio de otro programa.
- `int& durb` -Duración (en minutos) del bache.

```
Hora calcularBache(Hora hi1,int durp,Hora hi2,int& durb);
```

Se pide:

1. Generar el archivo `DESCARTADOS.dat`, con aquellos programas que, por cualquier motivo, fueron removidos de la propuesta de planificación. La causa será: 1 si el programa se superponía con otro, o 2 si se emitía dentro del horario ATP sin tener la calificación habilitante.
2. Generar el archivo `BACHES.dat` con los intervalos de tiempo que, luego de haber descartado los programas problemáticos, quedaron sin programación.

#### DESCARTADOS

```
struct Descartado
{
    int idPlanif;
    int causa;
};
```

#### BACHES

```
struct Bache
{
    int horaDesde;
    int minDesde;
    int duracion;
};
```

### 1.1.21. Medición de audiencia (4 y 5)

La empresa que se encarga de mediar el *ranking* de los programas de televisión dispone del archivo `MUESTRAS.dat`, con los datos del encendido, *zapping* y pagado que registraron los monitores de audiencia que se encuentran distribuidos en los puntos estratégicos de la ciudad. Este archivo no tiene orden, y por su tamaño puede ser administrarlo en memoria.



Análisis y solución

#### MUESTRAS

```
struct Muestra
{
```

#### MINAMIN

```
struct MinAMin
{
```



```
int idMonitor;
int hora;
int min;
char accion;
int canal;
};
```

```
int canal;
int min; // min del día (de 0 a 1439)
int cant;
};
```

El campo `accion` indica qué acción se ejecutó a la hora/minuto indicada por los campos `hora` y `min`. Esta acción puede ser:

- 'E' – Enciende.
- 'C' – Cambia de canal.
- 'A' – Apaga.

Las acciones 'E' y 'A' no involucran ningún canal, por lo que el campo `canal` no trae ninguna información válida en tales casos. La acción 'C' se produce cada vez que el usuario abandona un canal, habiéndolo dejado fijo por, al menos, 1 minuto.

Si luego de ver un determinado canal (durante más de 1 minuto) el usuario apaga el televisor, el monitor generará, primero, un registro con `accion = 'C'`, cuya hora/minuto coincidirán con la hora/minuto del evento de apagado.

Algo similar sucederá si la TV permanece encendida, en un mismo canal, de un día para el otro. Al llegar la hora 23:59:59 se generará un registro con `accion = 'C'`. luego se generará un evento de apagado y otro de encendido.

Se garantiza la consistencia de los datos. Es decir: para un monitor equis, existirá un registro con `accion = 'E'`. Luego habrá uno o varios registros 'C', y finalmente un registro 'A'. Esta secuencia podría repetirse varias veces, pues los televidentes miran TV muchas veces al día.

Se pide grabar el grabar el archivo `MINAMIN.dat` (Minuto a Minuto), con la estructura de registro presentada más arriba, ordenado por `canal` y `min`. Describiendo, por cada canal, cuántos monitores lo sintonizaron durante cada minuto del día. El campo `min` indica el minuto del día, cuyo valor estará comprendido entre 0 y 1439.

**1.1.22. Infracciones (4 y 5)**

Una municipalidad requiere procesar las actas de infracción labradas por sus inspectores de tránsito. Para esto cuenta con los siguientes archivos: `ACTAS.dat`, con el detalle de dichas actas, e `INFRACCIONES.dat`, con los diferentes tipos de infracción y sus respectivas penalidades.



Análisis y solución

**ACTAS**

```
struct Acta
{
    int idInspector;
    char pat[10];
    int fecha; // aaaammdd
    int hora; // hhmm
    int idInfraccion;
};
```

**INFRACCIONES**

```
struct Infraccion
{
    int idInfraccion;
    char descr[100];
    double penalidad; // $
    int diasPromo;
    double dtoPromo; // ej: 0.20
};
```

Cada tipo de infracción prevé un descuento por pronto pago. Es decir, si el infractor se presenta a pagar la multa antes de los `diasPromo` días, se le aplicará un descuento del `dtoPromo` por ciento sobre la `penalidad`. El campo `dtoPromo` contiene el coeficiente por el cual se debe multiplicar la `penalidad` para obtener el descuento a aplicar.

Se dispone del TAD `Fecha`, cuyas funciones (las que considere necesarias) se deben prototipar pero no desarrollar.

Se pide:

1. Por cada tipo de infracción, cantidad de infracciones labradas por día del mes.
2. Por cada patente, importe adeudado; según el siguiente listado:

Patente	Total Deuda	Total Descuento	Neto a pagar
XXXXXXXXXX	999999.99	999999.99	999999.99
:	:	:	:

### 1.1.23. Predios de fútbol (4 y 5)

Un estudio administrativo que gestiona predios con varios canchas de fútbol cada uno, requiere un programa que procese la reserva de canchas para el mes próximo.

Los clientes reservan una cancha en un predio para un determinado día del mes, en alguno de los tres turnos disponibles: *turno mañana* (de 10 a 14 hs.), *turno tarde* (de 14 a 18 hs.) y *turno noche* (de 18 a 22 hs.).



Análisis y solución

Disponemos de los siguientes archivos:

#### PREDIOS

```
struct Predio
{
    int idPredio;
    char nombre[100];
    char direccion[100];
    int cantCanchas;
    int idBarrio;
};
```

#### CANCHAS

```
struct Cancha
{
    int idPredio;
    int nroCancha;
    double precio;
    int flgCubierta;
    char obs[200];
};
```

#### RESERVAS

```
struct Reserva
{
    int idReserva;
    int idCliente;
    int idPredio;
    int diaMes; // 1 a 31
    char turno; // 'M', 'T', 'N'
    char celContacto[50];
};
```

#### RESERVAS (archivo a generar)

```
struct Rechazo
{
    Reserva reserva;
    int motivoRechazo;
};
```

La asignación de la cancha y la hora queda a criterio del programador. Es decir, el cliente sólo indica un predio y un turno. Qué cancha (dentro del predio) y qué hora (dentro del turno) lo establece el programa.

Se pide:

1. Generar el archivo `RECHAZOS.dat` con registros `Rechazo` para aquellas reservas que no serán aceptadas, por cualquiera de los siguientes motivos:
  - a. Cancha no disponible (`motivoRechazo=1`)
  - b. Predio inexistente (`motivoRechazo=2`)
2. Notificar a los clientes sobre la aceptación o rechazo de su reserva. En caso de rechazo, se debe informar el motivo del mismo. En caso de aceptación, informar el número de cancha y la hora (dentro del turno) asignada.
3. Emitir un listado detallando, para cada barrio, cuántas reservas fueron aceptadas y cuántas rechazadas por cada turno.