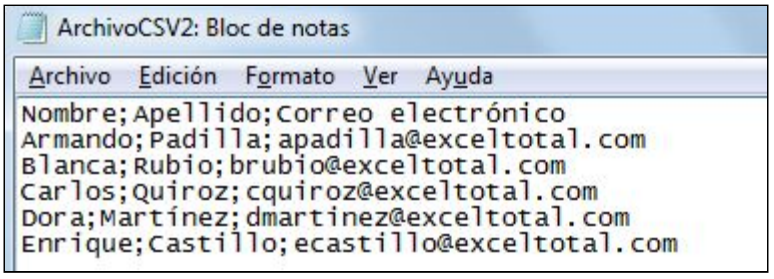
Tablas Dinámicas

PRÁCTICA FUNDAMENTOS DE PROGRAMACIÓN

Un fichero CSV (del inglés *'comma separated values'*) es un fichero de texto plano que contiene datos separados por comas. Por extensión, también se puede llamar CSV a los ficheros de texto con datos estructurados en forma de tabla que usan de separador cualquier otro carácter (espacio en blanco, tabulador, punto y coma, etc...), o conjunto de caracteres (por ejemplo, la doble almohadilla '**##**', el doble pipe '**||**' u otros). Un fichero CSV suele tener como primera línea (aunque no siempre) una cabecera (o header) para indicar el nombre de cada una de las columnas de datos del fichero. Ejemplos:



Fichero CSV con separador ';' y cabecera: **Nombre / Apellido / Correo**

Fichero CSV con separador '|' y sin cabecera

En esta práctica se van a considerar solamente ficheros CSV con cabecera, y cuyos datos, a partir de la segunda fila, serán de alguno de los siguientes tipos:

- NUM: datos numéricos, tanto enteros como reales
- DATE: fechas en formato AÑO/MES/DÍA (YYYY/MM/DD)
- STR: cadena de caracteres (abreviatura de '*string*')
- VOID: tipo vacío (se aplicará cuando haya un valor inexistente o nulo)

La cabecera (primera línea) de un fichero CSV indica el número de columnas de datos que tiene el fichero, si alguna de las filas siguientes a la cabecera no tuviera la misma cantidad de valores se considerará una fila errónea. Por otra parte, la primera fila de datos (la que sigue a la cabecera) indicará de qué tipo es cada una de las columnas del fichero.

También se considerarán filas erróneas aquellas que no tengan los mismos tipos de datos, y en el mismo orden, que la primera fila. Cuando se detecte una fila errónea en el fichero de datos deberá ser ignorada.

El programa a realizar debe ser capaz de leer un fichero CSV (con cabecera), que ha de contener un conjunto de datos estructurados en forma de tabla y, aplicando ciertos filtros y condiciones, realizar determinadas operaciones para obtener información más elaborada contenida en el fichero de datos. El programa deberá responder a diversos tipos de órdenes (comandos) que el usuario introducirá por consola, mediante los cuales se indicará al programa las operaciones que debe realizar.

El programa deberá mostrar un "***prompt***" que indique el nombre del fichero CSV que se esté utilizando para trabajar con él, por ejemplo, si se ha indicado un fichero de datos llamado "datos.csv" el prompt deberá mostrar un aspecto parecido a este:

`[datos.csv-?]:`

o este:

`[datos.csv]:`

La interrogación (-?) indica si el fichero ha sido validado o no (más adelante quedará mejor explicado este concepto). Cuando el programa está recién arrancado, y aún no se ha establecido un fichero, se indicará poniendo un asterisco en el prompt:

`[*]:`

A continuación se van a indicar los comandos que el programa deberá ser capaz de reconocer y ejecutar. El texto inscrito entre los caracteres 'menor' y 'mayor' (**< . . . >**) se refiere a un argumento o parámetro que será necesario incluir en el comando correspondiente. Si a su vez, un argumento aparece inscrito entre corchetes (**[< . . . >]**), eso querrá decir que dicho parámetro es opcional. Los comandos que debe ejecutar el programa son los siguientes:

`salir`

Comando para salir del programa (no lleva parámetros), produce la finalización del mismo. Se debe liberar la memoria ocupada (no dejar basura) y, sin mostrar por pantalla ningún tipo de mensaje, terminar su ejecución.

`datos <nombrefichero> <sep>`

Establece el fichero de datos indicado por el parámetro **`<nombrefichero>`** para las operaciones posteriores. Dicho fichero deberá ser un CSV con datos delimitados por el separador **`<sep>`**. De manera particular, si el parámetro **`<sep>`** fuera igual a

la cadena "**[tab]**"o la cadena "**[esp]**" (incluyendo los corchetes, tanto en mayúsculas como en minúsculas) se entenderá que el separador es un tabulador o un espacio en blanco respectivamente. El comando solo debe verificar si el fichero existe o no. Si el fichero existe simplemente se cambia el prompt incluyendo la interrogación (-? ⇒ fichero pendiente de validar). Si el fichero no existe se mostrará un mensaje por pantalla indicándolo, el prompt y el estado en general del programa quedará en este caso como estaba antes del comando. Cuando se establece un nuevo fichero de datos se libera toda la memoria que pudiera haber reservada a causa de las operaciones realizadas anteriormente.

validar

No tiene parámetros. Para el fichero establecido (cuyo nombre debe aparecer en el prompt), y considerando el separador que se indicó con el comando **datos**, se verifican cuantas filas y columnas contiene y de qué tipo son. En pantalla se irá mostrando un listado de las líneas erróneas indicando su nº (si las hubiera). Al final se mostrará el número de filas incorrectas, el número de filas correctas, el número de columnas y se enumerarán los nombres de dichas columnas, indicando de qué tipo es cada una. Dichos metadatos (filas, columnas y tipos) deberán almacenarse para ser usados posteriormente por otros comandos. Al validar un fichero se elimina la interrogación del prompt. Si previamente no se hubiera indicado un fichero de datos (con el comando **datos**), se indicará con un mensaje de error. Si se ejecuta este comando por segunda vez sobre el mismo fichero se vuelve a mostrar la información de metadatos almacenada, no se deberá leer el fichero por segunda vez (no se mostrarán en este caso las filas erróneas). OJO, un fichero CSV debe tener en su primera fila (cabecera) el mismo número de valores que en la segunda (datos), de no ser así la validación sería fallida, se indicará con un mensaje de error y el prompt no cambiará.

NOTA: *Los comandos que se muestran a continuación solo se podrán ejecutar si previamente se ha indicado un fichero de datos y éste ha sido validado. En caso de no darse esta circunstancia, no se podrán ejecutar y el programa lo deberá indicar con un mensaje de error sin hacer ninguna otra operación.*

renombrar <columna> <nuevacolumna>

Actualiza/modifica el nombre de una de las columnas del fichero validado, no directamente en el fichero, sino en los metadatos almacenados al validarlo. Si no existe la **<columna>** indicada o **<nuevacolumna>** ya existe (no puede haber dos columnas con el mismo nombre), se indica con un mensaje de error y no se hace nada más.

info <columna>

Recorre/lee el fichero validado y, considerando solamente las líneas correctas, muestra cierta información en pantalla sobre la **<columna>** indicada según sea el

tipo de dato de dicha columna (o un mensaje de error si **<columna>** no existe):

- NUM: Muestra el máximo, el mínimo, el promedio y calcula los recuentos para un histograma de 5 columnas.
- DATE: Muestra el máximo, el mínimo y calcula los recuentos para un histograma de 5 columnas.
- STR: Muestra un listado de cada valor de la columna indicando cuántas veces se repite dicho valor.

Esta información, una vez recopilada, pasa a ser parte de los metadatos almacenados en el programa. Si se ejecuta por segunda vez este comando sobre una misma columna, el programa deberá mostrar por pantalla la misma información, pero obtenida de los metadatos, no leyendo nuevamente el fichero.

`filtro <columna> <operador> <valor>`

Este comando establece un filtro o condición para operar con los datos. El argumento **<columna>** debe ser el nombre de una de las columnas del fichero de datos validado. El argumento **<operador>**, debe ser uno de los operadores relacionales tal y como se escriben en C (==, !=, >, >=, <, <=). Por último, el argumento **<valor>** es cualquier valor compatible con la columna indicada, es decir, del mismo tipo (numérico, fecha o cadena). Si hay algún error en los argumentos (p.ej.: la columna no existe, la columna y el valor no son del mismo tipo, ...) se indica con un mensaje de error, Si todo es correcto el filtro queda establecido, y se guarda en memoria.

`filtros`

(En plural) Enumera en pantalla, numerándolos de 1 en adelante, los diferentes filtros que se han establecido con el comando **filtro** (en singular) anterior. Si no hay filtros establecidos simplemente se indica con un mensaje.

`borrar <num>`

Borra el filtro número **<num>** de entre los que se han establecido previamente. El parámetro **<num>** debe ser un número entero positivo entre 1 y el número total de filtros que haya establecidos, si no se cumple esta condición se indica con un mensaje de error y no se borra ningún filtro.

`aplicar <opción> [<columna>]`

Aplica los filtros que se han establecido previamente, es decir, busca en el fichero de datos las filas correctas que cumplen con todas las condiciones indicadas en dichos filtros y realizan con ellas la **<opción>** indicada. A continuación se enumeran los posibles valores del argumento **<opción>** (en mayúsculas o minúsculas) y el resultado que deben generar y mostrar en pantalla:

- **CUENTA**: Hace un recuento de las filas correctas que cumplen con la condición de todos los filtros establecidos. Con esta opción el parámetro **<columna>** (opcional) no es necesario, será un error incluirlo, en tal caso se indicará con un mensaje de error y no se realizará ninguna operación.
- **TOTAL**: Calcula el total de la **<columna>** indicada, en este caso el segundo parámetros (**<columna>**) si es necesario, y debe ser de tipo NUM, en caso contrarío, se indicará con el correspondiente mensaje de error.
- **PROMEDIO**: Análogo al anterior, pero en vez de calcular una suma total, calcula el promedio de los valores que cumplen con la condición del filtro.

Nótese que la condición establecida por un filtro podría ser tan restrictiva que ninguna línea del conjunto de datos la cumpla, en un caso así el comando simplemente indicaría con un mensaje que "*No hay datos que cumplan el filtro*" (o algo similar). Por otra parte, si no se ha establecido ningún filtro se realizarán los cálculos sobre todas las líneas correctas del conjunto de datos.

Hasta aquí la descripción de los comandos que el programa debe poder interpretar y ejecutar. Nótese que, salvo los dos primeros comandos ("**salir**" y "**datos**") todos los demás requieren que previamente se haya establecido un fichero de datos con el que trabajar, de no ser así, se estará incurriendo en un error que deberá dar como resultado un mensaje en pantalla indicando la falta de un fichero de datos para poder ejecutar la acción. También son situaciones de error aquellas en las que un comando va seguido de una cantidad de parámetros incorrecta, distinta a la indicada en cada caso, o cuando los parámetros no son del tipo esperado (p.e.: texto en vez de un número). En estos casos el programa también deberá responder con un mensaje de error adecuado. Otras veces, el comando puede ser correcto, pero tal vez no se pueda ejecutar por diferentes motivos, por ejemplo, puede que el nombre del fichero indicado no exista, o que el nombre indicado de una columna tampoco exista en el conjunto de datos establecido; en situaciones de este tipo, el comando tampoco se podrá ejecutar y el programa deberá responder con un mensaje de error adecuado.

La estructura general del programa deberá seguir el siguiente esquema:

```
1. Imprimir en pantalla nombre, apellidos y e-mail del autor
2. Imprimir un 'prompt' y esperar a que el usuario escriba un
   comando por teclado (leer un string con 'gets()')
3. Analizar el contenido de la cadena introducida
  3.1. Si se corresponde con un comando incorrecto, se indica con
       un mensaje de error
  3.2. Sino, si se corresponde con un comando correcto pero este
       no se pudiera ejecutar, se indica con el mensaje más
       apropiado posible
  3.3. Sino, se ejecuta el comando según las especificaciones
4. VOLVER AL PASO 2 (excepto si el comando es "salir")
```

Obviamente, este pseudocódigo indica únicamente como debe ser el funcionamiento general del programa, pero no dice nada acerca de cómo deben ejecutarse los comandos introducidos por el usuario (paso 3.3 del algoritmo), diseñar y programar esos procedimientos es tarea del alumno.

IMPORTANTE*:** Explícitamente se exige que el programa no muestre por pantalla más texto ni mensajes de los indicados en el enunciado. Tampoco deberá hacer pausas del tipo "Pulse INTRO para continuar...***" ni deberá borrarse el contenido de la pantalla.

IMPLEMENTACIÓN: Detalles y normas de obligado cumplimiento

Para la representación y almacenamiento de los datos en memoria se deberán usar **OBLIGATORIAMENTE** los tipos de datos enumerados y las estructuras de datos que se van a describir a continuación:

Para representar los diferentes tipos de las columnas de un fichero de datos:

```
typedef enum {VOID, NUM, DATE, STR} TIPO;
```

Los metadatos de las columnas se almacenarán en nodos de una lista, para construir dicha lista se deben crear las siguientes estructuras:

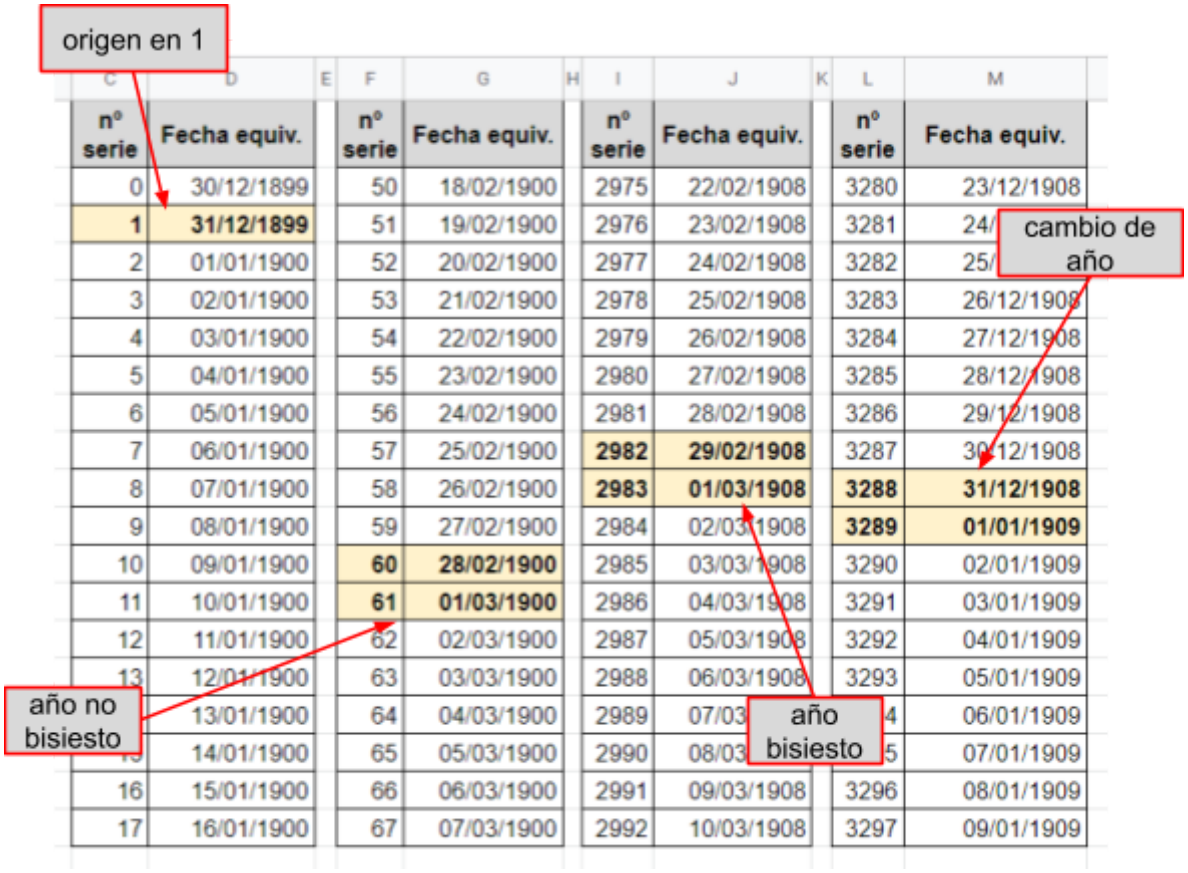
```
// Estructura: ETIQUETA
typedef struct etiq {
   char *etiqueta;          // nombre de la etiqueta (texto)
   int cuenta;              // veces que la etiqueta se repite
   struct etiq * siguiente; // puntero siguiente etiqueta
} ETIQUETA;

// Estructura: COLUMNA
typedef struct col {
   char *nom;             // nombre de la columna
   TIPO t;                // tipo de la columna
   float max, min, prom;  // máximo, mínimo y promedio
   int histograma[5];     // información para histograma
   ETIQUETA *lista;       // lista de etiquetas STR
   struct col *next;      // puntero a la siguiente columna
} COLUMNA;

// Estructura: METADATOS
typedef struct {
   int nErrs, nFils, nCols; // n° de errores, filas y columnas
   COLUMN *p;               // puntero a primera columna de datos
} METADATOS;
```

Con las estructuras arriba definidas se debe crear una lista enlazada de columnas, la estructura principal es la estructura **METADATOS**, esta estructura representa a la lista de columnas. Cada columna estará representada por una estructura (nodo) de tipo **COLUMNA** que contiene todos los elementos para almacenar los metadatos de una columna, independientemente del tipo que sea (**NUM**, **DATE**, **STR**, **VOID**). Ciertamente no todos los elementos de dicha estructura se usarán en todos los casos, por ejemplo, para las columnas de tipo **DATE**, no se calcula el promedio, la variable **prom**, queda sin uso en este caso. Las columnas de tipo **STR**, deben poder almacenar una lista de valores y un número que indique cuántas veces se repite dicho valor, el atributo **lista** (en la estructura **COLUMNA**) se utilizará en este caso para apuntar al primer nodo de una lista de estructuras (nodos) de tipo **ETIQUETA** que contendrá dicha información, ojo, solo cuando la columna sea de tipo **STR** (en el resto de casos **lista** quedará sin uso). En general, el alumno debe determinar cuándo usar o no usar alguna de las variables de las estructuras definidas.

Los elementos **max** y **min** (en **COLUMNA**), se utilizarán para guardar los valores máximo y mínimo cuando un dato sea de tipo **NUM**, pero también cuando sea de tipo **DATE** (fecha). Para almacenar una fecha como un número el alumno deberá programar algunas funciones que permitan convertir una fecha en un número de serie equivalente y viceversa. El concepto de número de serie aplicado a fechas consiste en contar los días que han transcurrido desde una fecha que se toma como origen hasta la fecha dada, ese recuento de días será el número de serie correspondiente a dicha fecha. Muchas hojas de cálculo usan ese tipo de conversiones para almacenar internamente el valor de una fecha. Véase como ejemplo la figura siguiente de una “Hoja de cálculo de Google”:



nº serie	Fecha equiv.	nº serie	Fecha equiv.	nº serie	Fecha equiv.	nº serie	Fecha equiv.
0	30/12/1899	50	18/02/1900	2975	22/02/1908	3280	23/12/1908
1	**31/12/1899**	51	19/02/1900	2976	23/02/1908	3281	24/
2	01/01/1900	52	20/02/1900	2977	24/02/1908	3282	25/
3	02/01/1900	53	21/02/1900	2978	25/02/1908	3283	26/12/1908
4	03/01/1900	54	22/02/1900	2979	26/02/1908	3284	27/12/1908
5	04/01/1900	55	23/02/1900	2980	27/02/1908	3285	28/12/1908
6	05/01/1900	56	24/02/1900	2981	28/02/1908	3286	29/12/1908
7	06/01/1900	57	25/02/1900	**2982**	**29/02/1908**	3287	30/12/1908
8	07/01/1900	58	26/02/1900	**2983**	**01/03/1908**	**3288**	**31/12/1908**
9	08/01/1900	59	27/02/1900	2984	02/03/1908	**3289**	**01/01/1909**
10	09/01/1900	**60**	**28/02/1900**	2985	03/03/1908	3290	02/01/1909
11	10/01/1900	**61**	**01/03/1900**	2986	04/03/1908	3291	03/01/1909
12	11/01/1900	62	02/03/1900	2987	05/03/1908	3292	04/01/1909
13	12/01/1900	63	03/03/1900	2988	06/03/1908	3293	05/01/1909
	13/01/1900	64	04/03/1900	2989	07/03	4	06/01/1909
	14/01/1900	65	05/03/1900	2990	08/03	5	07/01/1909
16	15/01/1900	66	06/03/1900	2991	09/03/1908	3296	08/01/1909
17	16/01/1900	67	07/03/1900	2992	10/03/1908	3297	09/01/1909

Para almacenar los filtros también es necesario crear una lista donde ir guardando o borrando, cuando corresponda, los filtros que se hayan definido. Para crear esta lista de filtros se definen los siguientes tipos de datos:

```
// Tipo enumerado: OPERANDO
typedef enum {
     IGUAL, DISTINTO, MENOR, MENORIGUAL, MAYOR, MAYORIGUAL
} OPERANDO;

// Estructura: COLUMNA
typedef struct fil {
   COLUMN * pCol;          // puntero a la columna
   OPERANDO operador;      // operación a realizar
   char *valor;            // valor a aplicar
   struct filtro *next;    // puntero al siguiente filtro
} FILTRO;

// Estructura: FILTROS
typedef struct {
   int num;    // nº de filtros contenidos en la lista
   FILTRO *p; // puntero al primer filtro
} FILTROS;
```

El tipo enumerado **OPERANDO** representa las diferentes operaciones que se pueden aplicar en un filtro. La estructura **FILTROS** (en plural) es análoga a **METADATOS**, representa la lista de filtros. Ahora, cada filtro se representa con una estructura (nodo) **FILTRO** (en singular) donde se almacena la información del filtro establecido, un puntero a la columna indicada, un entero para codificar el tipo de operador y el valor a aplicar en dicho filtro.

***IMPORTANTE*:** Con respecto a todos los tipos de datos definidos en este enunciado, queda a criterio del alumno, si lo considera necesario, añadir algún atributo adicional si lo encuentra oportuno (tanto en la práctica o en los futuros exámenes).

La utilización de estas estructuras de datos implica realizar gestión de memoria dinámica, reservando la memoria que se vaya a necesitar y liberándola cuando ya no sea necesaria. Cada vez que se ejecute un comando que requiera reservar memoria deberá comprobarse que efectivamente dicha memoria se ha reservado correctamente, en caso de que no sea así, el comando no se podrá ejecutar y el programa deberá mostrar en pantalla un mensaje indicando el error por falta de memoria.

En esta práctica hay que hacer un uso intensivo de varias formas de operaciones con cadenas de caracteres por lo que, para abordar la implementación de la aplicación, se necesita una librería de funciones que permitan hacer ciertas operaciones con dichas cadenas, a continuación se describe la funcionalidad de algunas de esas operaciones que

se recomienda implementar en forma de librería:

1. Una función que, dada una cadena de caracteres y un separador (que será otra cadena más corta), devuelva el número de campos que hay en dicha cadena.
2. Una función que, dada una cadena, un separador y un número 'i', devuelva el valor del campo i-ésimo contenido en esa cadena.
3. Una función que valide si una cadena tiene formato (o forma) de número entero válido con opción a que pueda empezar con un signo + (número positivo) o - (número negativo). También para número reales (con decimales).
4. Una función que devuelva el valor numérico de una cadena que representa un valor entero (o usar atoi()).
5. Una función que devuelva el valor numérico de una cadena que representa un valor real (o usar atof()).
6. Una función que elimine de una cadena los espacios en blanco, tabuladores '\n' y '\r' que pueda contener al principio y al final (ver función '*trim*' en otros lenguajes de programación, p.e., trim() en PHP)'.
7. Una función que determine si una cadena de caracteres representa una fecha válida.
8. Una función que a partir de una fecha válida devuelva su número de serie equivalente (y su opuesta).

La forma (prototipo) que deben tener estas funciones es algo secundario y queda a elección del alumno, lo importante es disponer de dichas funciones para abordar el resto del problema. Por convenio la librería que contenga dichas funciones deberá llamarse "**cadenas**", es decir, deberá estar formada por los ficheros "*cadenas.h*" y "*cadenas.c*".

Por otra parte, la práctica también debe contener la librería "**datos**" (ficheros "*datos.h*" y "*datos.c*"), para albergar las definiciones de tipos y estructuras descritos anteriormente, así como todas las funciones que el alumno necesite crear para manipular las listas que deben crearse en memoria.

Como en todo programa que manipula estructuras de memoria dinámica, es muy importante definir y programar funciones para construir y destruir dichas estructuras, es decir, para reservar y liberar memoria, así como para inicializarlas cuando son creadas.

PREGUNTAS / DUDAS SOBRE EL ENUNCIADO

Ejemplos de Archivos CSV

Hay infinidad de sitios en la web de donde se pueden descargar datos abiertos en formato CSV (y otros), os paso algunos links:

> Contrataciones en municipios de la comunidad valenciana:
> http://www.dadesobertes.gva.es/storage/f/file/20160413192341/contratos---municipios---genero----2016---01.csv
>
> Estaciones meteorológicas (enero 2017)
> http://www.dadesobertes.gva.es/storage/f/file/20160413143852/contaminacion-atmosferica-y-ozono---promedios-diarios---1997-01.csv
>
> Resultados elecciones locales 2015
> http://www.dadesobertes.gva.es/storage/f/file/20160414174803/resultados-locales----2015.csv

Para localizar esos ficheros me he basado en el catálogo de datos abiertos de http://datos.gob.es, filtrando por "CSV" y "Comunidad Valenciana", en el siguiente link hay más:
http://datos.gob.es/es/catalogo?publisher_display_name=Generalitat+Valenciana&res_format_label=CSV&_publisher_display_name_limit=0

MUY IMPORTANTE
Abrid los ficheros para ver cual es su separador, a veces es la coma (,) otras vece es el punto y coma (;), otras puede ser el tabulador (\t), etc... podría cambiar según el origen de datos.

--