Libro de ejercicios

Programación para Ingeniería Eléctrica

27 de febrero de 2018

NOTA:

Los ejercicios marcados con un asterisco (∗) son complementarios. Sirven para practicar más profundamente algunos conceptos, pero no son esenciales.
Los ejercicios marcados con una daga (†) son ejercicios complementarios asociados a obligatorios.

Práctico 1 - Introducción

Ejercicio 1 - Función módulo (MOD)

La función módulo, más conocida por su abreviación MOD, calcula el residuo de una operación de división entera, de modo que 7 MOD 3 = 1. Generalizando, $x\ MOD\ y$ lo podemos escribir como $x - k \cdot y$, donde k es el entero inferior más cercano del cociente x/y.

a) Escribir un programa que calcule el MOD de dos números y lo imprima en pantalla. Estos números serán fijados como constantes en el programa.

b) Escribir una función `mod` que tome dos enteros como parámetros y devuelva el módulo de ambos como resultado entero. Probarlo con un programa ejecutable como el que se muestra debajo (recuerde que la función a implementar tiene que estar definida antes del comienzo de `main()`).

```
#include <stdio.h>

int main()
{
  int m;
  m = mod(4,5);
  /* %d se rellena con el valor de m al imprimirse la cadena */
  printf("mod(4,5)=%d\n", m);
  return m;
}
```

Ejercicio 2 - Simulador de dados

Realizar un programa que simule la tirada de un dado, devolviendo un valor aleatorio entre 1 y 6 en cada invocación. Se sugiere investigar respecto a la función `rand()`[1] contenida en la librería estándar.

[1] *The C Programming Language* Kernighan & Ritchie - Apéndice B5

Ejercicio 3 - La Gaussiana

La función normal utilizada en aplicaciones estadísticas tiene la siguiente expresión:

$$y = \frac{1}{\sigma\sqrt{2\pi}}\, e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$

a) ¿Cuántos parámetros de entrada tiene esta función?

b) Realizar un programa que implemente la función normal, devolviendo el valor en punto flotante (`float`) de y resultante para cualquier combinación de sus parámetros de entrada.

c) Escriba un `main` que evalúe la función para los valores $\mu = 1$, $\sigma = 4$, $x = 2$ y lo imprima en pantalla. Verifique su resultado con una calculadora.

NOTA: *Se sugiere investigar en la bibliografía sugerida la librería <math.h>; la misma contiene las funciones `sqrt` y `exp`, y la constante numérica `MATH_PI`, que pueden resultar de gran utilidad. Además, para poder compilar exitosamente programas con estas funciones, es necesario agregar la opción `-lm` al final de la linea de comandos de compilación, ejemplo:*

```
$cc gauss.c -lm -o gauss
```

Ejercicio 4 - Verificando condiciones

Realizar un programa que calcule e imprima la raíz cuadrada y el inverso de un número fijo. Antes de calcular la raíz se debe verificar que el número no sea negativo y antes de calcular el inverso se debe verificar que el número no sea nulo. De no cumplirse estas condiciones el programa debe alertar al usuario de estos problemas mediante un texto en pantalla.

Ejercicio 5 - Argumentos desde la linea de comandos

Es posible capturar los argumentos desde la línea de comandos si la función `main` es declarada de modo que tome dos parámetros `argc` y `argv` (el tipo de datos `char **` de `argv` sera visto mas adelante):

```
#include <stdio.h>

int main(int argc, char **argv)
{
  int n;
  printf("Número de argumentos: %d\n",argc);
  n = 0;
  while (n < argc) {
    printf("Argumento %d vale %s\n", n, argv[n]);
    n++;
  }
  return n;
}
```

a) Compile y ejecute el código anterior para distintos parámetros cualesquiera.

En los clásicos descuentos del impuesto al valor agregado (IVA), hoy en día 22%, mucha gante comete el error de quitar el 22% del precio de plaza, olvidando que este es el resultado de agregarle el

22% al precio original sin la carga impositiva. De modo que $Precio\_de\_plaza = 1,22 \cdot Precio\_original$.

b) Realizar un programa que reciba el precio de plaza y retorne el precio original. Modificar el programa anterior de forma que el valor del IVA sea ingresado por el usuario, esto brindará robustez frente a cambios en la política impositiva.

Ejercicio 6 - Dígitos decimales de un número entero

Escriba un programa que imprima uno a uno los dígitos decimales de un número entero dado. Se sugiere utilizar la función `mod` vista en ejercicios anteriores.

Ejercicio 7 - Dígito verificador

La Dirección Nacional de Identificación Civil (DNIC) nos adjudica en la cédula de identidad un número único para cada habitante de nuestro país. Dicho número contiene un dígito verificador generado automáticamente. Dados los 7 dígitos de un número de cédula $c_0c_1c_2c_3c_4c_5c_6$, siendo c_0 los millones y c_6 las unidades, el algoritmo para calcular el dígito verificador v es el siguiente:

1. $s = 2c_0 + 9c_1 + 8c_2 + 7c_3 + 6c_4 + 3c_5 + 4c_6$
2. r es el menor múltiplo de 10 mayor a s
3. $v = r - s$

A modo de ejemplo, para el número de cédula 4213264 se tiene $s = 101$, luego $r = 110$, y finalmente $v = 9$.

a) Escriba una función `verif` que tome como argumentos los 7 dígitos de la cédula como un arreglo de enteros y devuelva el dígito de verificación como otro entero. Escriba un `main` para probarla con una o más cédulas conocidas fijas en el código (por ejemplo la suya).

b) Escriba un programa que reciba el número de cédula (sin verificación) desde la línea de comandos y lo muestre luego con su dígito verificador al final (tal como se muestra en la cédula, separado por un guión). Para convertir la cadena de texto ingresada a un número entero de 7 dígitos, utilice la función de la biblioteca estándar `atoi`, por ejemplo

```
int main(char argc, char **argv) {
  int numero_cedula;
  ...
  numero_cedula = atoi(argv[1]);
  ...
}
```

y luego utilice el algoritmo del Ejercicio 6 para extraer los dígitos y guardarlos en el arreglo que toma la función en la primera parte.

Ejercicio 8 - Calculador de propinas (*)

Realizar un programa de nombre `propina` que calcule propinas y divida la cuenta entre los comensales. El programa debe ser invocado desde la línea de comandos de la siguiente forma

```
$./propina comensales comida porcentaje
```

donde `comensales` es el número (entero) de comensales, `comida` el costo (sin decimales) de la comida, y `porcentaje` es un valor entero entre 0 y 100 indicando el porcentaje del costo de la comida que se aplicará como propina.

Como resultado, el programa debe mostrar a pantalla algo similar a lo siguiente

```
Propina total: xxxx
Propina por persona: xxxx
Precio total a pagar: xxxx
Precio a pagar por persona: xxxx
```

NOTA: *Se sugiere utilizar la función `atoi` para convertir una cadena de texto a entero. Recuerde que para poder acceder a los argumentos en la línea de comandos, la función main debe ser declarada como `int main(char argc, char** argv)`; luego el argumento i-ésimo puede accederse como cadena de texto como `argv[i]`.*

Práctico 2 - Tipos, Operadores y Expresiones

Ejercicio 1 - Tamaño a medida

Escriba un programa que tome un número entero como argumento desde la línea de comandos, e indique al usuario cuál es el tipo de entero más pequeño capaz de representarlo correctamente. Por ejemplo, el número 1022 no entra en un `char`, pero sí en un `int`. Se sugiere utilizar la función `atol` de `<stdlib.h>` para convertir el argumento de línea de comandos a un `long int` y luego verificar si cabe en tipos más pequeños, asignándolo a esos tipos y verificando que se preserva el valor. Una alternativa es medir la cantidad de bits necesaria para representar al número en cuestión (qué función matemática puede servir para ello?) y luego utilizar el operador `sizeof` para seleccionar el tipo que mejor calce.

Ejercicio 2 - Inicialización de variables automáticas

Compile y ejecute el siguiente código. Explique las diferencias observadas.

```
#include <stdio.h>

int k; /* variable global */

int f() {
  int a;
  a++;
  return a;
}

int g() {
  int a = 0;
  a++;
  return a;
}

main() {
        int i;
        printf("k=%d\n",k);
        for (i = 0; i < 10; i++) {
                printf("a en f() vale %d\n",f());
        }
        for (i = 0; i < 10; i++) {
                printf("a en g() vale %d\n",g());
        }
}
```

Ejercicio 3 - Operador módulo

El operador módulo entero `%` implementa en hardware la funcionalidad vista en la función `MOD` descrita en el práctico 1. Para probarlo, supongamos que quiere imprimir una guía para saber en qué columna

está una letra en la pantalla. Escriba un programa que pueda imprimir cadenas de caracteres de largo n arbitrario, formada por repeticiones concatenadas del patrón `‘‘+----’’`. Por ejemplo, para $n = 12$ la cadena sería `‘‘+----+----+--’’`. **El programa debe consistir en un sólo bucle; no vale dos!**

Ejercicio 4 - Buffer circular (†)

Los buffers circulares son estructuras de datos muy sencillas pero muy importantes en la programación de algoritmos relacionados con la ingeniería eléctrica, en particular de filtros digitales. Un buffer es, en general un espacio de almacenamiento temporal de datos de tipo “FIFO” (First In First Out), es decir, los datos se sacan del buffer en el orden en que fueron introducidos. Generalmente, un buffer consiste en un arreglo de un cierto largo máximo (llamémosle `NMAX`), un índice al último lugar que se escribió (llamémosle `iin`) y un índice al último lugar que se leyó (`iout`). Las operaciones que pueden realizarse sobre un buffer son: guardar un nuevo dato, y tomar el dato más viejo. Según la definición de los arreglos de C, el índice lineal más bajo en tal buffer es `0` y el más alto es `NMAX-1`.

Los buffers circulares tienen la particularidad de que los índices se consideran *módulo NMAX*, es decir, si uno intenta acceder a un buffer en la posición `NMAX+1`, el lugar en memoria que será leido es el del arreglo en la posición `1`. De la misma manera, si se intenta acceder a un índice negativo, por ejemplo `-1`, el lugar del arreglo que será devuelto es `NMAX-1`.

a) Considere un arreglo `buffer` de caracteres de largo `NMAX=4`, y un entero `i`

b) Escriba una o más expresiones en C para almacenar un nuevo valor `char a` en la posición `i+1` del buffer, incrementando en el proceso el valor de `i`.

c) Escriba una o más expresiones en C para *leer* el valor del buffer en la posición `i-k`, (sin actualizar `i`), donde `k` es un entero arbitrario.

d) Intente repetir las dos partes anteriores utilizando otras expresiones de C.

e) Al respecto de las dos partes anteriores, existe alguna ventaja en que `NMAX` sea una potencia de `2`? Qué podría hacerse en ese aso?

f) Escriba un programa que defina las variables anteriormente mencionadas, rellene el buffer `buffer` con el caracter `'X'`, inicialice `i=0`, y luego uno a uno, alimente los caracteres del primer argumento de la linea de comandos en el buffer y luego imprima el caracter del buffer en la posición `i-3`. (Debería obtenerse una salida “retardada” en donde los primeros tres caracteres impresos sean 'X'.)

Ejercicio 5 - Operadores lógicos

Escriba una expresión lógica que implemente la función lógica $z = f(w, x, y)$ dada por la siguiente tabla:

0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

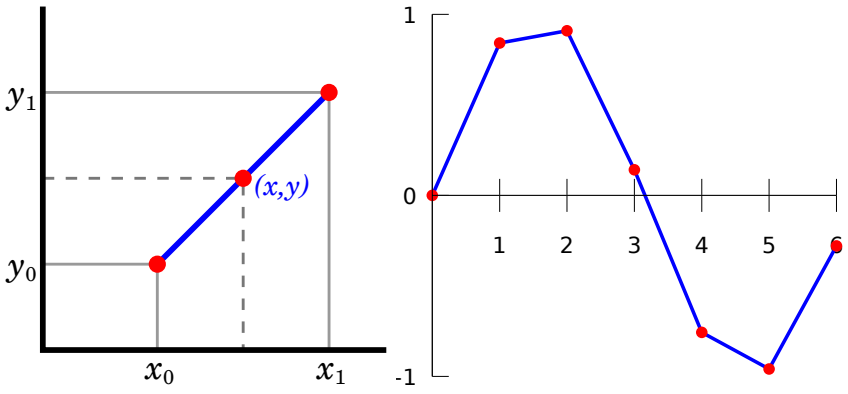

Figura 1: Interpolación lineal.

Ejercicio 6 - Operadores de bit

Los operadores de bit `|`,`&`,`^`,`~`,$\gg$,$\ll$ son muy imporantes en aplicaciones de bajo nivel. El siguiente ejercicio pone a prueba su uso con pequeños problemas.

a) Representación binaria Escriba un programa que tome un entero sin signo e imprima su representación en base binaria.

b) Enmascarado Escriba un programa que tome tres argumentos enteros. El primero es un entero cualquiera n, y los siguientes dos, llamémosles a y b, sirven para delimitar una máscara binaria que se usará para apagar todos los dígitos binarios del número n que estén por debajo de la posición a y a partir de la posición b inclusive, y luego imprima el resultado. Por ejemplo, si tenemos el número $n = 45$ (en binario 00101101), $a = 2$ y $b = 5$, el resultado sería 00001100.

c) Encriptado sencillo Escriba un programa que tome dos cadenas de texto: una larga, especificando un `texto` a encriptar, y una corta, a usar como `clave` de encriptación. La encriptación se realiza haciendo `xor` de **los primeros 6 bits** de cada letra en el `texto` a encriptar con una letra correspondiente de la `clave`. Al principio se alinean las dos cadenas de texto y cuando el índice en el texto original supera al de la clave, se retoma desde su comienzo. A modo de ejemplo, si el texto es "hasta la vista baby" y la clave es "terminator", la correspondencia es la siguiente:

```
HASTA LA VISTA, BABY!
TERMINATORTERMINATORT
```

y el resultado encriptado:

```
\DAYH.MU/D]VFL%.CUMK5
```

Si todo sale bien, el texto encriptado resultante (que puede contener caracteres no imprimibles) puede ser recuperado aplicando la misma función sobre él, usando la misma clave.

Nota: Se sugiere primero resolver el problema de manejo de bits y luego atacar el resto. A modo de ayuda, recuerde que puede especificar constantes numéricas en base hexadecimal poniendo como sufijo `0x`. Por ejemplo, `0x7f` es el decimal `127`, binario `01111111`.

Ejercicio 7 - Interpolación de funciones y LUTs (†)

Una de las relaciones de compromiso más comunes en la ingeniería, cuando se trata de implementar software, es *memoria vs. velocidad*. Uno de los ejemplos más clásicos de esto son las *tablas de funciones* o *LUT* (del inglés Look-Up Table). La idea es tener almacenados en memoria los valores

de una función $f(x)$ costosa de calcular (por ejemplo, $\cos(x)$), de modo que aproximar su valor mediante esa tabla resulte mucho menos costoso que calcularlo de cero. En general, la tabla almacena $f(x)$ para un conjunto de N valores consecutivos de x dentro de cierto rango, por ejemplo $\mathcal{X} = \{0, \delta, 2\delta, \ldots, (N-1)\delta\}$. Valores de $f(x)$ fuera de ese conjunto deberán ser *interpolados* en base a valores de $f(x)$ que sí estén almacenados.

En este punto, surgen varias alternativas: Por un lado, para un mismo tamaño de tabla, se obtendrán mejores aproximaciones si se utilizan técnicas de interpolación más sofisticadas. La técnica más burda es el valor más cercano (devolver $f(\hat{x})$ tal que $|x - \hat{x}|$ es mínimo). En la otra punta tenemos interpolación cúbica, polinómica, o Splines. Por otro lado, si se fija un método de interpolación, la aproximación mejorará a medida que la tabla sea más "fina" (es decir, δ sea menor, y la tabla tenga más puntos). Para un mismo rango a representar, esto claramente requiere más memoria.

La idea de este ejercicio es la de implementar distintas variantes de LUTs, con distintos métodos de interpolación, y ver su resultado en términos de error cuadrático.

a) Implemente un programa que reciba como argumento un entero N y luego:

1. declara un arreglo de valores de punto flotante de nombre `lut` y tamaño `N`
2. Rellene dicho arreglo con un período de la función `cos(x)`, de modo que el primer elemento de `lut` contenga el valor de $\cos(0)$ y el último $\cos(2\pi(N-1)/N)$. (se sugiere aproximar π como $\text{acos}(-1.)$.) Cuál es el valor de δ en este caso?
3. Estime el siguiente error de aproximación (aproximando la integral como una suma de Riemann en pasos $\Delta x = 2\pi/10000$),

$$e = \int_{x=0}^{x=2\pi} (cos(x) - c\bar{o}s(x))^2 dx,$$

   donde $c\bar{o}s(x)$ es la aproximación dada por la interpolación de vecinos más cercanos en la tabla,

$$c\bar{o}s(x) = \text{lut}[\hat{i}],\ \hat{i} = \arg\min_i \{|x - i\delta| : i = 0, 1, \ldots, N-1\},$$

   es decir, el valor más cercano a x en el conjunto $\mathcal{X}$.
4. Imprima el valor de N, el tipo de interpolación usada (en este caso vecino más próximo) y el valor de e.

Ejecute el programa anterior para varios valores de N, observando su salida.

b) Repita el ejercicio anterior utilizando interpolación lineal. En este caso

$$c\bar{o}s(x) = [\hat{x}],\ \hat{x} = (x_1 - x)\text{lut}[x_0] + \text{lut}[x_1](x - x_0),$$

donde

$$x_0 = \text{lut}[\hat{i_0}],\ \hat{i}_0 = \arg\max_i \{i\delta \leq x, i = 0, 1, \ldots, N-1\}$$
$$x_1 = \text{lut}[\hat{i_1}],\ \hat{i}_1 = \arg\min_i \{i\delta > x \in, i = 0, 1, \ldots, N-1\},$$

es decir, los dos enteros mas cercanos a x (uno por arriba, el otro por abajo) en la tabla.

Ejercicio 8 - Frecuencias, histogramas y probabilidades empíricas (*)

Posiblemente la forma más sencilla de estimar probabilidades es a través de frecuencias empíricas, es decir, el conteo de ocurrencias de los distintos eventos aleatorios posibles a lo largo de un número grande de repeticiones de un cierto experimento.

Supongamos que en experimento hay m posibles eventos, y que la ocurrencia de cada evento es representada como un entero W entre 0 y $m-1$. Nuestros datos de entrada consisten en un vector de largo n, $\mathbf{x}$, donde $x_i = j$ indica que el experimento i-ésimo dió como resultado $W = j$.

Definamos como $f_j, 0 \leq j < m$ a la cantidad de veces que el evento $W = j$ ocurrió en $\mathbf{x}$. Al vector de valores $\mathbf{f} = (f_0, f_b, \ldots, f_{m-1})$ lo denominamos *histograma*. Dado el histograma histograma $\mathbf{f}$, la probabilidad del evento $W = j$ puede ser estimada como $P(W = j) = f_j/n$.

a) Escriba una función `int calcular_histograma(int x[], int m, int n, int f[])` que tome una serie de n datos en `x` con valores entre 0 y $m-1$, y calcule las frecuencias de cada uno de los posibles m eventos en el arreglo de largo m, `f`.

b) Complete el siguiente programa (el archivo **C** correspondiente se llama `histograma.c` y está disponible en la página del curso) con la función que usted acaba de escribir, ejecútelo y lea con atención la salida del programa.

```
#include <stdio.h>

#define N 10
#define M 3
int numeritos[N] = {0,1,2,1,2,0,0,1,2,2};

void calcular_histograma(int datos[], int m, int n, int f[]) {
  /* RELLENAR AQUI */
}

int main() {
  int i;
  int f[M];   /* frecuencias */
  float P[M]; /* probabilidades */

  calcular_histograma(numeritos,M,N,f);

  for (i = 0; i < M; ++i) {
    P[i] = f[i]/N;
    printf("frecuencia f[%d]=%d -> probabilidad P[%d]=%f\n",i,f[i],i,P[i]);
  }
}
```

c) Verifique que las frecuencias fueron bien calculadas (debería ser $f_0 = 3$, $f_1 = 3$, $f_2 = 4$). ¿Qué sucede con las probabilidades estimadas? ¿Qué está mal? Modifique el programa anterior para que las probabilidades se calculen correctamente.

d) Modifique su función `calcular_histograma` de modo que si llegara a ocurrir que $x_i > m-1$ (para el cual no está previsto registrar la ocurrencia), se incremente en su lugar la frecuencia del evento $W = m-1$. De la misma manera, si llegara a ocurrir que $x_i < 0$, es la frecuencia del evento $W = 0$ la que debe incrementarse. Esta estrategia es conocida como "clipping" o "recorte", y puede resumirse

de la siguiente manera:

$$y = \text{clip}_{a,b}(x) = \begin{cases} a & , \quad x < a \\ x & , \quad a \leq x \leq b \\ b & , \quad b < x \end{cases}$$

e) Desafío intente escribir la función $\text{clip}_{a,b}(x)$ en una sóla línea, usando el operador ternario de asignación condicional `?:`. Este operador es en general mucho más eficiente que utilizar sentencias `if-else`.

Ejercicio 9 - Representación de matrices

Hay esencialmente dos maneras de representar una matriz bidimensional $\mathbf{X}$ de tamaño $m \times n$ en **C**. La más intuitiva es definirla como un arreglo bidimensional

```
float X[m][n];
```

Una manera alternativa, menos intuitiva pero muchas veces más conveniente, es la de definir dicha matriz en un arreglo unidimiensional de largo mn,

```
float X[mn];
```

y definir una correspondencia entre los índices bidimensionales de la matriz $(i, j), 0 \leq i < m, 0 \leq j < n$ y los índices unidimiensionales $0 \leq k < mn$. Una manera posible de hacer esto último es "por filas", es decir, los primeros n elementos de `X` (`X[0]` a `X[n-1]` inclusive) corresponden a la primera fila de la matriz $\mathbf{X}$, los siguientes n elementos ((`X[n]` a `X[2n-1]` inclusive) a la segunda fila , etc.

Esta es la forma en que por ejemplo se suelen almacenar imágenes digitales en memoria para su posterior procesamiento.

a) Implemente una función `double elemento(double X[],int i, int j)` que devuelva el elemento (i, j) de una matriz bidimensional almacenada por filas en un arreglo unidimiensional `X`.

b) (*) Modifique la función anterior de modo que si se especifican coordenadas fuera de rango, éstas sean recortadas mediante la función clip descrita en el ejercicio anterior, es decir que el valor devuelto por la función sea $\mathbf{X}(i', j')$ donde $i' = \text{clip}_{0,m}(i)$ y $j' = \text{clip}_{0,n}(j)$. (Esto se utiliza mucho en procesamiento de imágenes para "estirar los bordes" de la imagen).

Práctico 3 - Control de Flujo/Manejo de cadenas

Ejercicio 1 - While versus for

Escriba el programa descrito en la parte (a) del ejercicio 6 del práctico 2 (imprimir un número entero en binario) utilizando:

a) Un ciclo `for`

b) Un ciclo `while`

c) El ciclo `do-while`

Ejercicio 2 - Multiplicación de matrices

Escriba un programa que calcule el producto $\mathbf{y} = \mathbf{A}\mathbf{x}$ entre una matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$, almacenada en un arreglo bidimensional `double A[m][n]`, y un vector $\mathbf{x} \in R^n$, almacenado en un arreglo `double x[n]`, y lo almacene en un vector `double y[n]`. Los operandos pueden ser fijos en el programa; no tienen por qué ingresarse en la línea de comandos.

Ejercicio 3 - Generación de ondas (†)

a) Escriba un programa de nombre `tri.c` que tome como argumento un valor entero N y un valor real A calcule e imprima en pantalla la secuencia de valores $x(n)$, con $0 \leq n < N$, definida de la siguiente manera:

$$x(n) = \begin{cases} -A + 4(A/N)n & , \quad 0 \leq n < N/2 \\ A - 4(A/N)(n - N/2) & , \quad N/2 \leq n < N \end{cases}$$

La secuencia debe ser impresa de a un valor por linea de la salida, junto con su argumento, es decir, cada linea debe tener la forma

```
n       x(n)
```

b) Redirija la salida del programa hacia un archivo de texto **tri.txt** mediante la siguiente sintaxis:

```
./tri N > tri.txt
```

el > en el comando anterior es el operador de "redirección de salida" de UNIX. Eso indica que la salida, en lugar de ir a la terminal, va a un archivo de texto. También puede redirigirse salidas hacia las entradas de otros programas. Esto lo veremos bien al final del curso.

c) Plotee el resultado utilizando el comando **graph** del paquete GNU Plotutils (`apt-get install plotutils`):

```
graph -T png -g 3 < tri.txt > tri.png
```

notar el comando anterior utiliza redirección de entrada < y salida >. El de salida fue comentado en la parte anterior. El de entrada hace que el contenido de **tri.txt** sea usado como "entrada" al comando **graph** en lugar de lo usual (el teclado). El resultado es una imagen de tipo `PNG` con una sencilla gráfica de la función generada.

d) Repita las partes anteriores para el caso $x(n) = \cos(\phi_n)$, con $\phi_n = \frac{2\pi}{N} n$.

Ejercicio 4 - Detección de un pulso

Un problema clásico en comunicaciones es la detección de un *pulso* en una señal $x(t)$ que es recibida por ejemplo desde una antena. La figura **??** muestra un ejemplo de un pulso en una señal eléctrica que toma voltajes entre 0 y $5V$. Para la detección, la señal es observada en intervalos temporales de duración T (supongamos por ejemplo $T = 10^{-3}s$), y se decide que ha habido un pulso si se suceden un número n (supongamos $n = 5$) de observaciones consecutivas durante las cuales el voltaje medido supera un cierto umbral u, en este caso $u = 2.5V$.

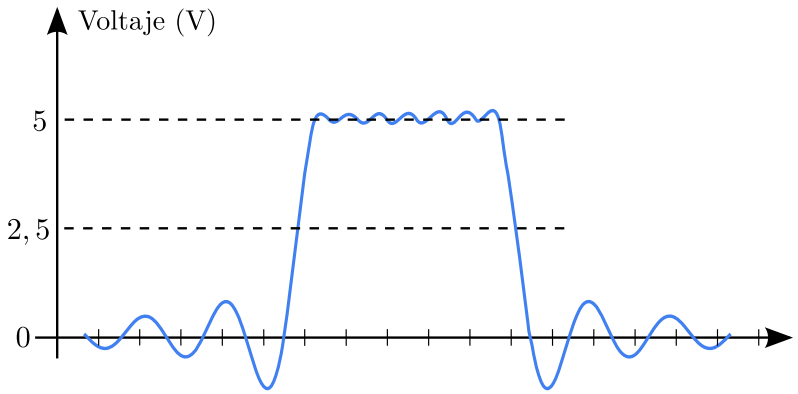

Figura 2: Ejemplo de pulso en una señal eléctrica. Las marcas horizontales corresponden a los momentos en que se mide el voltaje. En este caso, el pulso comienza luego de $t = 5T$ y termina luego de $t = 11T$.

a) Escriba una función `int detector(double x[], int m, double umbral, int n)` donde `x[k]` contiene la medida en el tiempo $t = kT$, `m` indica el tamaño del arreglo `double x[]`, `umbral` es un umbral de detección, y `n` es la cantidad mínima de muestras consecutivas por encima del umbral para decidir que ha ocurrido un umbral. La función debe imprimir a pantalla el caracter `'0'` mientras no ocurra un pulso, y el caracter `'1'` desde el momento en que se detecta un pulso, y durante la duración de éste. La función debe devolver `0` si no se detectó ningún pulso en `double x[]`, o `1` de lo contrario.

b) Simule y verifique el correcto funcionamiento de la función implementada con los siguientes datos

```
double x[] = {0.1,2.3,2.2,3.1,2.2,2.8,3.5,3.7,3.4,2.8,0.1,0.8,0.1,0.1,0.3,0.3,0.4,0.5,0.2};
umbral = 2.5;
n = 5;
```

Ejercicio 5 - Switch y máquinas de estados

La sentencia `switch` es muy útil a la hora de implementar *máquinas de estados*. Una máquina de estados (finita) tiene un estado interno s que puede tomar uno de un número finito de N valores posibles; representemos a este conjunto como $\mathcal{X} = \{s_1, s_2, \ldots, s_N\}$. La máquina recibe símbolos (por ejemplo bits, o letras) desde una entrada x que puede tomar valores en un conjunto $\mathcal{A} = \{\alpha_1, \ldots, \alpha_M\}$, de a uno a la vez.

Asumamos que la máquina comienza, antes de haber recibido ningún símbolo, con un estado $s(0)$. Al recibir el símbolo $x(i)$ en el instante $i > 0$, la máquina cambiará de estado de acuerdo al estado actual $s(i)$ y a $x(i)$, a través de una función de *transición de estados* $f : \mathcal{X} \times \mathcal{A} \to \mathcal{X}$, $s(i+1) = f(s(i), x(i))$.

La figura **??** muestra una forma de representar una máquina de estados finitos. En ese ejemplo, si el estado en tiempo i es $s(i) = s_1$ y el valor de entrada es $x(i) = 0$, el estado al recibir el símbolo

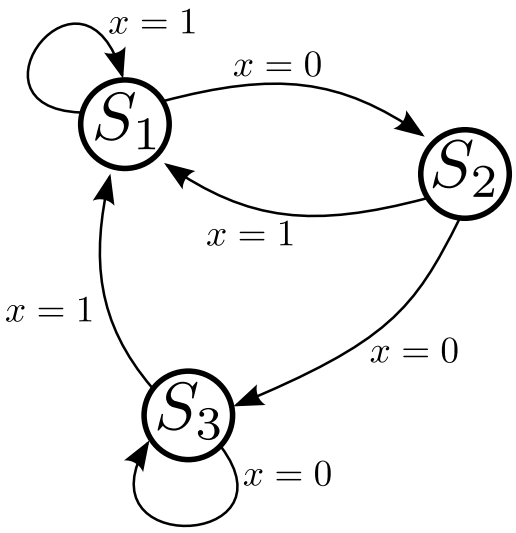

Figura 3: Máquina de estados con 3 estados $\mathcal{M} = \{s_1, s_2, s_3\}$ y dos posibles entradas $\mathcal{X} = \{0, 1\}$

.

$i+1$-ésimo será $s(i+1) = s_2$. Si por el contrario recibe $x(i) = 1$, el próximo estado será $s(i+1) = s_1$. **En adelante, representaremos al estado s_j con el valor entero j.**

a) Utilice una sentencia `switch` para implementar la función de transición de estados de la figura **??** como una función en C
`int proximo_estado(int si, int xi)`,
donde `si` representa a $s(i)$ y `xi` representa a $x(i)$.

b) Implemente un programa que simule la máquina de la figura **??** y muestre en pantalla secuencia de estados $s(i)$ resultante dado el estado inicial $s(0) = 1$ y la siguiente secuencia de entradas, representada como un arreglo de enteros:

```
int x[] = {0,1,1,0,0,1,1,1,0,0,0,1,0,0,0,0,0,1,1};
```

c) Reescriba la función de transición de estados con sentencias `if-else`.

d) Implemente el problema de detección de pulsos como una máquina de estados.

Ejercicio 6 - Break y continue

Las palabras `break` y `continue` suelen ser usadas para simplificar la lógica de cierto tipo de operaciones. Por ejemplo, `break` se usa mucho dentro de ciclos `for` o `while` para terminar prematuramente si alguna condición deja de cumplirse (la alternativa es mover esa condición a la declaración del loop, pero a veces eso es engorroso, sobre todo si hay muchas formas de salirse). Por su parte, `continue` es útil para *saltearse* partes de, por ejemplo, una cadena, que no interesa analizar. Los siguientes son dos ejemplos (no particularmente buenos, pero ilustrativos) de lo anterior:

```
/* devuelve 1 si los textos son idénticos hasta el caracter n-ésimo, o 0 en otro caso */
int comparar_texto(char s[], char t[], unsigned int n) {
  int i;
  int iguales = 1;
  for (i = 0; i < n; i++) {
    if (s[i] != t[i]) {
```

```
      iguales = 0;
      break; /* ya no interesa seguir mirando */
    }
  }
  return iguales;
}

/* calcula el producto interno entre los vectores v y w de largo n
double producto_interno(double v[], double w[], unsigned int n) {
  double a = 0.0;
  for (i = 0; i < n; i++) {
    if ((v[i] == 0.0)||(w[i] == 0.0)) {
      continue;
    }
    a += v[i]*w[i];
  }
  return a;
}
```

a) Escriba la función `comparar_texto` anterior **sin** usar `break`.

b) Escriba la función `producto_interno` anterior **sin** usar `continue`, pero sí evitando actualizar `a` cuando uno de los dos componentes `v[i]` o `w[i]` sea 0.

Ejercicio 7 - Comparación de cadenas

a) Escriba una función `interpretar_comando` que reciba una cadena de texto `char com[]` y que, según su contenido, imprima un texto distinto. Más específicamente:

- Si el contenido de `com` es "saludame", el programa debe imprimir "Hola."
- Si el contenido de `com` es "despedime", el programa imprime "Hasta luego."
- Si el contenido de `com` es "adulame", el programa imprime "Eres una maravilla"
- Si el contenido de `com` es "insultame", el programa imprime "Eres una basura despreciable"
- Si el contenido de `com` no es ninguno de los de arriba, el programa debe imprimir "No te entiendo."

La idea aquí es utilizar la función `strcmp` para verificar el contenido de `com`.

Ejercicio 8 - Control remoto (*)

Un ejemplo práctico de máquinas de estados es el de un receptor de control remoto de televisor. El control remoto envía los distintos comandos como secuencias ("trenes") de pulsos intermitentes de luz infraroja. Cuando uno no hace nada, el control no envía nada, pero cuando se presiona un botón, se genera un tren de pulsos particular al botón. El tren de pulsos se divide en general en tres tramos. El primero, llamado "secuencia de arranque", es una secuencia fácil de detectar que permite al televisor ponerse "alerta" a un comando que va a llegar. Luego viene el comando en sí, que consiste en una secuencia de bits particular, y luego viene una "secuencia de parada", que marca el fin del comando.

El receptor de infrarrojos actúa como una máquina de estados, donde los estados pueden ser 3: esperando un comando, iniciando comando, y decodificando un comando. En el primero, se está a la espera por la llegada de algún bit. En el segundo, se recibió uno o más bits de la secuencia de arranque y se está verificando que ésta sea una secuencia válida (es necesario sincronizarse al largo exacto de la secuencia de arranque para saber dónde comienza el primer bit de comando), y luego se pasa a decodificar el comando. Si no se identifica correctamente la secuencia de inicio, se pasa nuevamente al estado `ESPERA`.

Más concretamente, representaremos el estado del receptor con una variable `estado` que puede tomar tres valores posibles: `ESPERA=1`, `INICIO=2` y `COMANDO=3`.

- La secuencia de inicio es `01010101`.
- Los comandos posibles son `10000001` para "subir volumen", `10000000` para "bajar volumen", `10000011` para "mute/unmute", `00xxxxxx` para "cambiar a canal xxxxxx", donde `xxxxxx` es una palabra de 6 bits capaz de representar los canales 0 a 63.
- Los bits recibidos a partir de cierto tiempo inicial están almacenados en un arreglo de enteros `unsigned char x[n]` donde cada elemento puede valer `0` ó `1`.

a) Implemente una máquina de estados que represente al control remoto y decodifique los comandos de una secuencia de bits recibida. Para esto debe escribir una función `nuevo_estado`, que reciba como parámetros un entero corto sin signo `estado_actual` y un entero corto `x` representando un nuevo bit de la señal, y devuelva el nuevo estado del control remoto.

b) Escriba un programa que use la función anterior para decodificar los comandos de un control remoto. Al finalizar un comando válido, el programa debe imprimir el comando en cuestión y su parámetro. Por ejemplo, si se recibió la secuencia `00000101`, se podría imprimir algo como "CANAL 5". Note que para lograr esto debe llevar la cuenta, de alguna manera, de los bits que se van recibiendo desde el momento en que comienza el comando, hasta que termina el comando, y luego interpretar la cadena binaria de manera acorde.

Práctico 4 - Funciones y la estructura del programa

Ejercicio 1 - Funciones con memoria

Implemente la función `proximo_estado` del ejercicio "Switch y máquinas de estados" del Práctico 3, pero esta vez de modo que reciba sólo un parámetro, la nueva entrada `xi`. El estado interno de la máquina de estados debe ser almacenado de modo que se mantenga su valor entre llamadas sucesivas a la función, pero que dicho valor sólo sea visible desde adentro de la función `proximo_estado`. El valor inicial del estado será siempre `1`.

Ejercicio 2 - Visibilidad y alcance

a) Corrija el siguiente programa para que funcione (**tritri.c**). Considere cómo lo haría si, irremediablemente, la función `fun` tiene que ser definida *luego* del `main`.

```
int main() {
  return fun(10);
}

float fun(int x) {
  return 8*x;
}
```

b) Considere el programa **lolo.c**. Modifíquelo de modo que compile correctamente.

```
int a = 5;

int main() {
  int  b = 3;
  return a + b + c;
}

int c;
```

c) Descomprima el archivo **pepe.zip**, compile sus dos archivos fuente a archivos *objeto* (extensión `.o`), y obtenga una lista de los símbolos de cada uno utilizando la herramienta **nm**.

d) Genere un ejecutable de nombre **pepe** a partir de los archivos fuente de **pepe.zip**. Modifique sólamente el archivo **pepe_main.c** para que la compilación sea exitosa, y se imprima el contenido de la variable `a` definida en el archivo **pepe.c**.

e) Descomprima el archivo **coco.zip** y compile su contenido mediante la herramienta **make**. Analice el contenido de los archivos objeto mediante la herramienta **nm**. Modifique sólamente el archivo **coco.c** para que la compilación sea exitosa.

NOTA: *Recuerde que si tiene dudas acerca del uso de un cierto comando Unix, puede consultar su documentación mediante el comando* ***man****, por ejemplo* ***man nm****. Los archivos* **tritri.c**, **lolo.c**, **pepe.zip** *y* **coco.zip** *están disponibles para su descarga desde la página del curso.*

Ejercicio 3 - La herramienta make

Descomprima y estudie el contenido del paquete **lele.zip**, disponible para su descarga desde la página del curso. Escriba un archivo **Makefile** que genere un ejecutable de nombre **lele** a partir de esos tres.

Ejercicio 4 - Modularización

Considere un programa que deba cumplir las siguientes funciones. Asuma que cada función es suficientemente complicada como para requerir unas cuantas decenas, o incluso centenas, de líneas de código.

- Interpretar los argumentos de la línea de comandos y configurar el funcionamiento del programa
- Imprimir una descripción del programa razonablemente larga en caso de que no se utilice correctamente
- De acuerdo a lo ingresado en la línea de comandos, el programa puede realizar 4 acciones distintas: `comando_uno`, `comando_dos`, `comando_tres` y `comando_cuatro`. Cada acción es de por sí un pequeño programa.
- Cada comando tiene variables propias sobre las cuales opera, que no tienen por qué ser visibles al resto del programa.
- Cada comando a su vez puede ser dividido en tres etapas: *iniciar*, *ejecutar* y *finalizar*.

Plantee una posible organización del programa en funciones (sólo nombres, sin parámetros) y archivos fuente. De una lista de los archivos fuente y encabezados en los que dividiría el programa. De una lista de funciones a definir y declarar en los archivos fuente y encabezados anteriormente descritos.

Práctico 5 - Punteros, arreglos y memoria

Ejercicio 1 - Punteros y aritmética de punteros

Supongamos, como en el último ejercicio del práctico 3, que representamos a una matriz cualquiera $\mathbf{M}$ de tamaño $m \times n$ como un arreglo unidimensional `int* M` de tamaño `m*n`, donde cada bloque contiguo de tamaño n corresponde a una fila de $\mathbf{M}$, siendo los primeros n elementos (de 0 a $n-1$) los elementos de la primera fila, y así por delante.

a) Escriba una función `double traza(double *M, size_t n)` que calcule la traza de una matriz cuadrada $\mathbf{M}$ de tamaño $n \times n$. Intente hacerlo de la manera más eficiente posible, *sin* utilizar el operador `[]`, sólo con aritmética de punteros.

b) Escriba un programa que, dada una matriz $\mathbf{A}$ dos vectores $\mathbf{x}$ y $\mathbf{y}$, y dos escalares α, β, actualice $\mathbf{y}$ de modo que $\mathbf{y} = \alpha\mathbf{A}\mathbf{x} + \beta\mathbf{y}$. Intente hacerlo de la manera más eficiente posible.

c) Intente escribir la función anterior utilizando sólamente el operador `++`, buscando usar la menor cantidad posible de ellos.

Ejercicio 2 - Pasaje por referencia

Existen varias maneras de realizar el pasaje de variables como argumentos de una función. En el pasaje por valor se realiza una copia de la variable en otra dirección de memoria. Cada una de las variables, inicialmente con el mismo valor, pueden ser modificadas sin alterar a la otra. En el pasaje por referencia, en cambio, lo que se pasa es la dirección de memoria en donde está almacenado el dato. En este caso no se tienen 2 variables independientes, sino que se tiene un único valor referenciado (o apuntado) desde dos puntos diferentes. Cualquier acción que se realiza sobre el parámetro, se realiza en el mismo lugar de memoria que la variable pasada como parámetro.

a) Construir una función `rotar` que reciba cuatro enteros `a`, `b`, `c` y `d` y rote sus valores de manera que, al salir de la función, el valor de `a` quede almacenado en `b`, el de `b` en `c` y asi sucesivamente. Imprimir a consola desde el `main` los valores de `a`, `b`, `c` y `d` antes y después de llamar a la función rotar. Imprimir los valores también desde adentro de dicha función.

b) Para la función anterior probar qué pasa en ambos casos: pasaje por valor y pasaje por referencia.

c) Supongamos que se necesita realizar calculos dentro de una función y que el resultado de la función se almacena en más de una variable ¿cómo deben pasarse los argumentos, por valor o por referencia?

Ejercicio 3 - Opciones y argumentos en la línea de comandos

Este ejercicio muestra cómo se maneja típicamente una línea de comandos. Los argumentos en las líneas de comandos se separan (por convención) entre *opciones* y *parámetros*. Las últimas son los datos que en última instancia serán procesados por el programa invocado, por ejemplo, el nombre de un archivo. Las opciones modifican el comportamiento del programa. Un buen ejemplo es el del **gcc**. Veamos la siguiente línea:

```
$gcc -c -Wall -ansi -o pepe pepe.c
```

En este caso, `-c`, `-Wall`, `-ansi` y `-o pepe` son opciones, y el parámetro es `pepe.c`. Notar que las primeras tres opciones involucran un sólo argumento de la linea de comandos cada una (`argv[1]`, `argv[2]`, `argv[3]` respectivamente), mientras que `-o pepe` abarca dos argumentos, `argv[4]` y `argv[5]`. La lógica que define qué es un parámetro es algo que debe ser definida por el propio programador. En el caso anterior, la aparición de `-o` implica que el siguiente elemento en `argv` es un parámetro de la propia opción. En este caso, se trata del nombre del archivo de salida que `gcc` generará al copilar `pepe.c`.

Al describir el uso de un programa. Los argumentos obligatorios se suelen escribir entre <>, y los opcionales entre []. Por ejemplo:

```
./copiar [-o archivo_de_salida] <archivo_de_entrada>
```

indicaría que el programa `copiar` tiene como argumento obligatorio al nombre de un archivo de entrada, y que, opcionalmente, se puede especificar el nombre del archivo de salida con el argumento (compuesto) `-o archivo_de_salida`.

a) Escriba un programa que indique, para cada argumento de la línea de comandos, si es una opción o un parámetro (asumiremos que todas las opciones abarcan un sólo argumento).

b) Escriba un programa `contar` que reciba como parámetros dos números reales obligatorios `min`, `max`, un parámetro opcional `paso`, y una opción `-i`. El programa debe imprimir los números entre `min` y `max`, en incrementos de tamaño `paso`. Por defecto, si no se especifica `paso`, el incremento es `1`. Si aparece la opción `-i`, los números serán impresos de `max` a `min`, en decrementos de tamaño `paso`.

Si el programa no recibe suficientes argumentos (menos de 2), o si alguno de ellos es erróneo (por ejemplo, no son números válidos, o la opción no es reconocida), debe imprimirse el siguiente texto a pantalla:

```
Uso: contar [-i] min max [paso]
```

Ejercicio 4 - Cadena de caracteres

Escribir una función de tipo `void` que reciba dos argumentos: un entero (que será utilizado como entrada a la función) y un puntero a `char` (que será utlizado como salida de la función). La función debe evaluar si el entero es par o impar y devolver en el otro argumento las cadenas de caracteres “par” o “impar” según corresponda. Imprimir desde el main el mensaje devuelto por la función.

Ejercicio 5 - Arreglos de caracteres y sus funciones

En este ejercicio se trabajará con las funciones para manipulacion de cadenas de caracteres provistas por la libreria estandard.

a) Analizar el funcionamiento de las funciones `strstr`, `strrstr`, `strchr` y `strrchr`.

b) Considere la celebre frase del maestro Yoda definida como la variable `char* yoda = ‘‘Cuando mires al lado oscuro, cuidado debes tener... ya que el lado oscuro te mira tambien.’’`. Implemente una función que imprima en consola el texto de la variable anterior a partir de la palabra “mira”.

c) Para la variable `yoda` de la parte anterior, implemete una función que imprima en consola la primera parte de la frase (hasta los tres puntos suspensivos), luego un salto de linea, y luego la segunda parte de la frase. Los puntos suspensivos no deben aparecer impresos.

d) Considere la siguiente ruta a una imagen:
/home/pie/documents/practicos/5/imagenes/imagen.de.prueba.pgm.
Implemente una función que imprima en consola el nombre de la imagen sin la extension.

e) Para la ruta de la parte anterior implemente una función que imprima en consola toda la estructura de carpetas donde se encuentra la imagen, con un salto de linea entra cada directorio.

f) Sabiendo que dentro del directorio "practicos" hay un directorio para cada entrega cuyo nombre es el numero de entrega (como se muestra en la ruta de ejemplo), imprima en consola el numero del practico extraido de la ruta de ejemplo.

Ejercicio 6 - Punteros a funciones

a) Investigue el uso de la función `qsort` de `<stdlib.h>`. Utilice `qsort` para ordenar un arreglo de números dado.

b) Declare una variable de tipo *puntero a función que recibe dos enteros y devuelve un entero*

c) Escriba una función que tome tres arreglos `int *op1`, `int *op2` e `int *res`, todos ellos de largo `n`; y el puntero a una función como la descrita en la parte anterior `f`, de modo que al finalizar se cumpla `res[i]=f(op[i],op[i])` para todo $0 \leq i < n$.

Ejercicio 7 - Unique

La función `unique` recibe un arreglo de enteros `vIn` y su largo `lIn` y devuelve un arreglo de salida `vOut` y su largo `lOut`. El arreglo de salida contiene todos los elementos del arreglo de entrada pero sin repeticiones. Notar que no se sabe a priori cuál será el largo del arreglo de salida. Una forma de implentar esta función consiste en ordenar el arreglo de entrada e ir evaluando si un elemento es igual al anterior.

a) La primera parte consiste en calcular solamente el largo del arreglo de salida. Implementar una función `void` que reciba el arreglo de entrada y su largo y devuelva solamente el largo del arreglo de salida. Imprimir a consola el valor de `lOut` desde el `main`.

b) Modificar la función anterior para que también se devuelva el arreglo de salida. La firma de la función debe ser `void unique(int* vIn, int lIn, int** vOut, int* lOut)`. Imprimir a consola el valor de `lOut` y los valores de `vOut` desde el `main`.

c) Realizar un mapa de cómo quedan almacenadas las variables en la memoria.

Práctico 6 - Uso avanzado de punteros

Ejercicio 1 - Memoria dinámica

a) Escriba un programa `mem` que reserve un bloque de memoria dinámica para almacenar 1000000 enteros usando `malloc`, luego asigne el valor 0 a todos esos enteros mediante un ciclo `for`, y finalmente libere la memoria utilizando `free`. Mida el tiempo de ejecución utilizando el comando `time` de linux (lea su manual con `man time` para entender la salida):

```
\$ time ./mem
```

b) Escriba un programa como el anterior pero que en lugar de `malloc` y un `for`, utilice la función `calloc` para reservar e inicializar los elementos a 0 automáticamente.

c) Escriba un programa que reserve 10 enteros con `malloc`, luego les asigne los valores 1 a 10, luego invoque `realloc` para agrandar el espacio reservado a 20, imprima los valores de esos 20 elementos, y finalmente invoque a `free` para liberar el espacio reservado.

d) Supongamos que asigna el bloque de memoria reservado en las partes anterioes a un puntero de nombre `p`. Pruebe escribir fuera del array con, por ejemplo, `p[-1]`, `p[-10]`, `p[21]`, a ver qué sucede.

e) Pruebe invocar `free` sobre el mismo puntero dos veces consecutivas en alguno de los programas anteriores.

f) Pruebe invocar `free` sobre un puntero de valor `NULL` en alguno de los programas anteriores.

Ejercicio 2 - Destrozar el stack

Escriba una función `fun` que declare un arreglo `int x[10]` y luego escriba fuera de él, por ejemplo `x[-3]=10`. Escriba una función `main` que invoque a `fun`, ejecute el programa y observe qué sucede. Repita con distintos valores fuera de rango.

Ejercicio 3 - Punteros a punteros: matrices esparsas

La forma más sencilla de almacenar una matriz de m filas por n columnas es mediante un bloque contiguo de memoria de $m \times n$ elementos del tipo deseado (por ejemplo `double`).

Sin embargo, este tipo de almacenamiento es muy ineficiente, tanto desde el punto de vista de memoria como de costo computacional, cuando se trata de representar matrices *esparsas*, que son aquellas matrices donde la mayoría de sus elementos son 0. Este tipo de matrices ocurren muy a menudo en aplicaciones reales.

Para este caso, existen numerosas alternativas de almacenamiento. En este ejercicio veremos una de ellas.

A diferencia de las matrices comunes, el espacio que ocupan las matrices esparsas es variable, dependiendo de cuántos valores distintos de 0 hay en ellas. Por eso, en general, es necesario trabajar con memoria dinámica.

Una forma sencilla es representar cada elemento no nulo como un triplete (fila,columna,valor). Para esto podemos definir tres arreglos `int *fila`, `int *columna` e `double* val` y un entero `nnz` que lleva la cuenta de la cantidad de elementos no nulos, es decir, el largo de esos tres arreglos.

a) Implemente un módulo `esparso1.c` con las siguientes funciones:

- `void setval(int** pfila, int** pcolumna, double** pvalor, int* pnnz, int i, int j, int v)`: toma los datos de la matriz a modificar *por referecia* y los modifica de modo que el elemento (i, j) pase a valer v. Si el elemento modificado no estaba entre los elementos no nulos anteriormente (es decir, si no existe `k` tal que `fila[k]=i` y `columna[k]=j`, es necesario agrandar los arreglos con `realloc`, y agregar el nuevo elemento al final en los tres arreglos.
- `double getval(int *file, int* columna, double* valor, int nnz, int i, int j)`: obtiene el valor en la posicion (i, j) de la matriz especificada por los primeros cuatro valores. Si no se encuentra dicho elemento representado por los arreglos fila y columna, el valor devuelto es `0`.

Note que no es necesario especificar el largo y el ancho de la matriz para realizar las operacines anterioes. Escriba un programa para probar las funciones anteriores, por ejemplo, creando una matriz esparsa, populando algunos valores distintos de cero, y luego imprimiendo su contenido.

b) Claramente, agrandar los arreglos con `realloc` cada vez que se agrega un sólo elemento es lento y peligroso porque puede fragmentear mucho la memoria. Una alternativa muy usada es que los arreglos tengan un tamaño `nzmax` mayor a `nnz`. En este caso, sólo deberá agrandarse los arreglos cundo `nnz` supere a `nzmax`. Más aún, el nuevo valor de `nzmax` puede ser incrementado de a saltos mayores que 1, para evitar futuras llamadas a `realloc`.

- Agregue una función `void crear_matriz_esparsa(int nzmax_inicial, int** pfila, int **pcolumna, double** pvalor, int *nnz)` que inicialice adecuadamente a cada uno de los componentes de la matriz en base al valor de `nzmax_inicial`.
- Agregue una función `void agrandar_matriz_esparsa(int nuevo_nzmax, int** pfila, int **pcolumna, double** pvalor, int *nnz)` que aumente el tamaño de cada uno de los componentes de la matriz en base al valor de `nuevo_nzmax`.
- Modifique `setval`, de modo que reciba una referencia (puntero) a la variable `nzmax`, de modo que ésta también pueda ser modificada. Si `nnz` alcanza a `nzmax`, debe incrementarse el tamaño de los arreglos (y `nzmax`) en `10` unidades.

Ejercicio 4 - Funciones avanzadas de cadenas de texto

a) Estudie el uso de la función `strtok`. Utilícela para inicializar un vector de largo arbitrario especificado en la linea de comandos mediante la siguiente sintaxis (el programa se llama `main`):

```
\$./main -vector=1,2,3.2e4,4.1,-5.0,6.1
```

El programa debería imprimir la suma de las entradas del vector, y luego terminar. Si se reserva memoria dinámica (lo cual es buena idea porque no se sabe el largo del vector a priori), recuerde liberar la memoria.

b) Repita el programa anterior utilizando la función `strtod`.

c) Repita el programa utilizando la función `sscanf`.

Práctico 7 - Estructuras de datos

Ejercicio 1 - Tipos definidos

Definir los siguientes tipos:

- `real_t` como tipo `double`
- `vec_t` como puntero a elementos de tipo `real_t`
- `largo_t` como un entero largo sin signo

Ejercicio 2 - Uso básico de struct, union, enum

a) Defina un `struct` de nombre `vector` con dos campos: un puntero a punto flotante de precisión simple de nombre `datos`, y un entero largo sin signo de nombre `n`.

b) Defina al tipo `vector_t` en base a la estructura anterior.

c) Defina un `enum` de nombre codigos_de_error con tres elementos: `OK`, `ARCHIVO_NO_EXISTE`, y `FORMATO_INVALIDO`

d) Defina una `union` que, en el mismo lugar de memoria, represente tanto un `unsigned int` como un arreglo de 3 `char`.

e) Agregue, a la unión de la parte anterior, un campo de bits (`bitfield`) con tres campos: `rojo`, `azul` y `verde`, todos de 8 bits.

Ejercicio 3 - Estructuras y funciones

Consideremos la siguiente estructura:

```
struct estadistica{
  unsigned cuenta;
  int suma;
};
typedef struct estadistica estadistica_t;
```

Y la siguiente función:

```
void actualizar_estadistica (estadistica_t est, int valor){
  est.suma+=valor;
  est.cuenta++;
}
```

a) ¿Qué imprime el siguiente programa?

```

void main(){
  estadistica_t stats = {0,0};
  actualizar_estadistica(stats, 5);
  printf("estadisticas = {cuenta=%d, suma=%d}\n", stats.cuenta, stats.suma);
}
```

b) ¿Cómo modificaría la función `actualizar_estadistica` y la llamada a la misma para que se comporte como se espera?

Ejercicio 4 - Uniones y memoria

a) Considere las siguientes declaraciones:

```
enum tipo {CADENA, ENTERO};
struct estructura{
  enum tipo t;
  union u {
    int entero;
    char * cadena;
  } valor;
};
```

Donde se sabe que:

- sizeof(enum tipo) = 4
- sizeof(int) = 4
- sizeof(char *) = 8

¿Cúantos bytes necesita el compilador cómo mínimo para almacenar un elemento de tipo "struct estructura"? Justifique.

b) Verifique lo anterior escribiendo un programa que imprima los tamaños de los distintos tipos mencionados en la parte anterior.

Ejercicio 5 -Ñúmeros complejos

a) Definir una estructura capaz de representar un número complejo cuyas partes real e imaginaria se almacenen como números flotantes de precisión doble. Definir dicha estructura como `complejo_t` .

b) Declarar la estructura anterior en una archivo **complejo.h** junto con las siguientes funciones:

suma, resta, producto, cociente : todas reciben dos complejos y devuelven el resultado de la operación.

argumento : toma un complejo y devuelve su argumento en radianes como un flotante de doble precisión.

modulo : toma un complejo y devuelve su módulo como un flotante de doble precisión.

conjugado : Toma un complejo y devuelve su conjugado.

imprimir : Toma un complejo y lo muestra en pantalla como $x + y * j$ donde x e y son las partes reales e imaginarias respectivamente.

c) Implementar las funciones en un archivo **complejo.c** y probarlas en un `main.c`

Ejercicio 6 - Tipos de datos abstractos (*)

Una lista es un conjunto de elementos en el cual importa el orden y pueden haber elementos repetidos. Es decir, no es lo mismo la lista de enteros {1, 2, 3} que la lista {2, 1, 3}.

Una lista se puede implementar mediante el uso de un arreglo como por ejemplo el tipo `vector_t` definido en el ejercicio 2. Donde el primer elemento de la lista está ubicado en la posición 0 del arreglo, el segundo en la posición 1, etc. El incoveniente de esta implementación es que si se quiere agregar un elemento en una posición intermedia del arreglo hay que mover todos los elementos que siguen a esa posición.

Otra forma de implementar una lista de elementos es una lista enlazada, donde cada elemento se guarda en un nodo que tiene el valor del elemento y un puntero al siguiente nodo, o NULL si es el último de la lista.

a) Defina un `struct nodo` con dos campos:

1. Un campo de nombre `elemento` y de tipo entero simple;
2. Otro de nombre `siguiente` que sea un puntero a la propia estructura.

b) Defina el tipo `lista_t` como puntero a la estructura de la parte anterior.

c) Modifique la primera parte para definir el campo `siguiente` en función de el tipo `lista_t`

d) Escriba un archivo **lista.h** que contenga la definición del tipo `lista_t` y las siguientes funciones:

1. Una función **crear** que no tome parámetros y devuelva un elemento de tipo `lista_t`.
2. Una función **destruir** que reciba un elemento de tipo `lista_t`.
3. Una función **insertar** que tome como argumentos: un elemento del tipo `lista_t` llamado `lista`, un entero llamado `posicion` y un entero llamado `valor` y que retorne un entero.
4. Una función **eliminar** que tome como argumentos: un elemento del tipo `lista_t` llamado `lista` y entero llamado `posicion` y que no retorne nada.
5. Una función **buscar**, que tome una lista y un entero y devuelva la posición del elemento en la lista.
6. Una funcion **agregar**, que tome como argumentos una lista `lista` y un entero `valor` y devuelva un entero.
7. Una funcion **tamanio**, que tome una lista y devuelva su tamaño.

e) Implemente en un archivo **lista.c** las funciones declaradas en la parte anterior.

crear : debe crear una lista vacía.

destruir : debe liberar la memoria ocupada por la lista.

insertar : debe insertar el entero `valor` en la posición `posicion` de la lista `lista`, si la lista tiene al menos (`posicion` − 1) elementos. En este caso debe devolver 1 y en caso contrario 0.

eliminar : debe eliminar de la `lista` el elemento que está en la posición `posicion`, si la lista tiene al menos `posicion` elementos. Devolver 1 si lo pudo hacer 0 en caso contrario.

buscar : debe buscar la primera aparición de `valor` en la lista y retornarla, en caso de no encontrarlo devolver -1.

agregar : debe agregar el elemento `valor` al final de la lista. Devolver 1 si todo sale bien, 0 en caso de error.

tamanio : debe devolver la cantidad de elementos de la lista.

NOTA: *Observar que para poder representar la lista vacía el primer elemento debe ser ignorado. No participa en la búsqueda ni en la cuenta de tamaño. Cuando se elimina un elemento de la lista se debe liberar la memoria ocupada por el nodo.*

f) Implementar las operaciones `agregar`, `tamanio` y `destruir` de forma recursiva.

g) Escribir un programa que pruebe las funciones anteriores.

Ejercicio 7 - Tipos de funciones (*)

Considere la siguiente definición general de una función de densidad de probabilidades (PDF):

$$f(\mathbf{x};\theta),\ \mathbf{x} \in \mathbb{R}^n,\ \theta \in \mathbb{R}^k$$

El primer vector es el punto en $\mathbb{R}^n$ en donde se desea evaluar la distribución, y el segundo vector $\theta \in \mathbb{R}^k$ es el vector de parámetros de la distribución. Por ejemplo, para una gaussiana unidimensional se tendría $n = 1$ y $k = 2$, con $\theta = (\mu, \sigma^2)$.

a) Utilizando la definición del tipo `vector_t` del ejercicio anterior, defina un tipo `pdf_t` que sirva de puntero a funciones con la forma de $f(\mathbf{x};\theta)$, es decir, que tome dos parámetros de tipo `vector_t`: uno para describir $\mathbf{x}$ y otro para θ, y devuelva un valor de tipo `double` con el valor de la densidad en el punto $\mathbf{x}$.

b) Aquí veremos cómo usar el tipo definido anteriormente para implementar funciones que puedan operar con cualquier función de densidad de probabilidades.

En particular, vamos a escribir una función que aproxime la integral de una distribución de probabilidad mediante el método de Riemann, en una bola de radio r centrada en un punto $\mathbf{x}_0$, cubriendo una grilla de lado d, todos estos datos dados como parámetros. La función debe tomar pues los siguientes argumentos:

1. La función a integrar, de tipo `pdf_t`
2. El vector de parámetros θ como un parámetro de tipo `vector_t`
3. El centro de la bola, $\mathbf{x}_0$, como un parámetro de tipo `vector_t`
4. El radio de la bola, de tipo `double`
5. El espaciado de la grilla, de tipo `double`

y devolver un valor de tipo `double` con el resultado de la integral.

NOTA: *Para simplificar la implementación, asuma que $n = 1$. ¿Puede implementarlo para $n = 2$? Se le ocurre cómo podría hacer para implementarlo para n genérico?*

Práctico 8 - Entrada salida y manipulación de archivos

Ejercicio 1 - Hola archivo

a) Escriba un programa que abra el archivo **/tmp/hola.txt**, para escritura y luego lo cierre.

1. Ejecute el programa sin haber creado dicho archivo.
2. Verifique si el archivo existe luego de la ejecución.
3. Edite el archivo, escriba "Hola, archivo" y sálvelo.
4. Vuelva a ejecutar el programa.
5. Vuelva a abrir el archivo.

b) Modifique el programa de la parte anterior para que escriba "Hola, archivo" en el archivo **/tmp/hola.txt**.

Ejercicio 2 - wipe

Cuando eliminamos un archivo, el sistema operativo simplemente marca el espacio que ocupaba como libre o mejor dicho elimina la entrada en alguna tabla, según la implementación del sistema de archivos. No hace nada con el contenido del archivo.

En algunos casos si tenemos datos sensibles almacenados en un archivo, como contraseñas u otros datos privados, que queremos asegurarnos que se eliminen, debemos asegurarnos de sobreescribir los datos antes de borrarlo.

a) Escriba un programa **wipe** que reciba como parámetro un archivo y sobreescriba su contenido. Debe escribir el caracter `'0'` en cada posición del archivo pasado como parámetro. Si el archivo no existe el programa debe desplegar un error y no crearlo.

NOTA: *En la práctica el programa* **wipe** *además borraría el archivo y en realidad podría escribir, o bien algo aleatorio, o bien el byte* `'\0'`*; pero para poder verificar fácilmente su funcionamiento escribiremos el caracter* `'0'` *en ASCII. De esta forma podemos realizar la verificación con un editor de texto.*

Para verificar el funcionamiento del programa:

- Crear un archivo de texto y escribir algún contenido en el, por ejemplo "destruime".
- ejecutar el programa.
- verificar reabriendo el archivo y ver que tiene la misma cantidad de caracteres que el original pero sustituidos por el caracter `'0'`.

b) Luego de verificar que lo anterior funciona, agregar la eliminación del archivo al final del programa.

NOTA: *Para este ejercicio conviene tratar al archivo como binario y claramente hay que abrirlo en modo de actualización. Se sugiere ver las operaciones de la biblioteca estándar descritas en el apéndice B1 del libro del curso, en particular las 1.1, 1.4, 1.5, 1.6 y 1.7.*

En los medios magnéticos no alcanza con escribir una sola vez los datos, un programa serio de wipe, realizaría varias escrituras con diferentes patrones para efectivamente borrar todo rastro del archivo original.

Ejercicio 3 - Lectura/escritura de matrices

Escriba un programa que reciba tres parámetros, cada uno es una ruta de archivo. Los dos primeros contienen la definición de matrices a multiplicar, y en el tercero se debe escribir el resultado. Si el tercero no está presente se debe escribir el resultado en la salida estándar.

El formato de los archivos es simplemente una fila por línea, y dentro de la fila los elementos están separados por uno o mas caracteres en blanco o tabuladores.

Todas las filas deben contener la misma cantidad de elementos, en caso contrario se reportará un error.

Ejercicio 4 - Escritura y lectura de números en formato litle endian

Hay dos maneras de almacenar de manera binaria un entero de mas de un byte de largo. Se pueden guardar primero los bytes mas significativos o primero los menos significativos. A la primera se le llama big endian y a la segunda little endian.

El formato de audio WAV por ejemplo utiliza la convención little endian para almacenar los enteros de más de un byte.

Por ejemplo para un entero de 16 bits (dos bytes) se almacenan primero el byte menos significativo y luego el byte mas significativo.

Para un entero de 4 bytes por ejemplo $i = d*2^{24} + c*2^{16} + b*2^8 + a$ en un archivo se almacenarían en el orden a, b, c, d

a) Escriba una función `leer_entero_16` y una función `leer_entero_32` que reciban un puntero a archivo (`FILE *`) y devuelvan un entero.

La primera debe leer un entero de 16 bits en formato little endian de la siguiente posición del archivo. O sea debe consumir dos bytes, el primero será el byte menos significativo y el segundo el más significativo.

La segunda debe hacer lo mismo pero debe leer un entero de 32 bits o sea leer 4 bytes del archivo.

b) Escriba las funciones `escribir_entero_16` y `escribir_entero_32` análogas a las de la primera parte pero que escriban enteros en un archivo en lugar de leer. Dichas funciones recibirán dos parámetros, un puntero a archivo y el entero a escribir. Si el entero tiene mas de la cantidad de bytes a escribir, los más significativos se descartan.

c) Escriba un programa que pruebe dichas funciones haciendo lo siguiente:

- Reciba tres parámetros, un nombre de archivo, y dos enteros.
- Escriba en el archivo ambos enteros, el primero como de 32 bits y el segundo como de 16, utilizando las funciones implementadas en la segunda parte.
- Cierre el archivo y lo vuelva a abrir y lea ambos enteros del archivo utilizando las funciones de la primera parte.
- Imprima ambos enteros en pantalla para verificar que son iguales que lso que se ingresaron en la línea de comandos.

d) Utilice algún programa de edición de archivos binarios (como por ejemplo el programa bless) para comprobar que los enteros se guardaron cómo se esperaba.

NOTA: *El programa bless se puede instalar con el comando* `sudo apt-get install bless`

Ejercicio 5 - Imágenes color (obligatorio 2014) (*)

a) Escriba una función `leer_pixel_color` que reciba un puntero a archivo, devuelva un entero y realice lo siguiente:

- Lea tres bytes del archivo.
- Devuelva un entero cuyo byte mas significativo sea nulo, el segundo contenga el primer byte leído, el tercero contenga el segundo byte leído y el menos significativo el último byte leído.
- En caso de error la función debe retornar `-1`, que es un valor no posible. Por ejemplo si se acaba el archivo antes de leer el tercer byte.

b) Escriba una función `leer_pixels_color` que reciba los siguientes parámetros y devuelva un entero:

- Puntero a archivo `pf`
- Array de enteros `destino`
- Un entero `size`

La función debe:

- Leer `size` pixels del archivo referido por `fp`, en el formato de la función anterior.
- Colocar el i-ésimo píxel leído en la i-ésima posición de `destino`.
- Retornar 0 si se pudieron leer `size` píxeles correctamente y 1 en caso contrario.

c) Escriba una función `leer_encabezado` que reciba un puntero a un archivo y lea el encabezado de un archivo pnm. (Ver apéndice de la letra del obligatorio)

d) Escriba las funciones análogas a las de las primeras partes pero que escriban en lugar de leer.

e) Escriba un programa que pruebe las funciones de las partes anteriores, leyendo y escribiendo una imagen de ejemplo.

Ejercicio 6 - Graficando en ASCII

a) Escriba un programa que reciba como parámetros dos rutas de archivo y un flotante.

- El primer parámetro es un archivo que contiene un vector de números reales separados por caracteres de espacio, tabulador o salto de linea. Si no está presente se lee de la entrada estándar.
- El segundo parámetro es la ruta de un archivo de salida. Si no está presente se escribe en la salida estándar.
- El tercer parámetro es un coeficiente que si no está presente se considera 1.

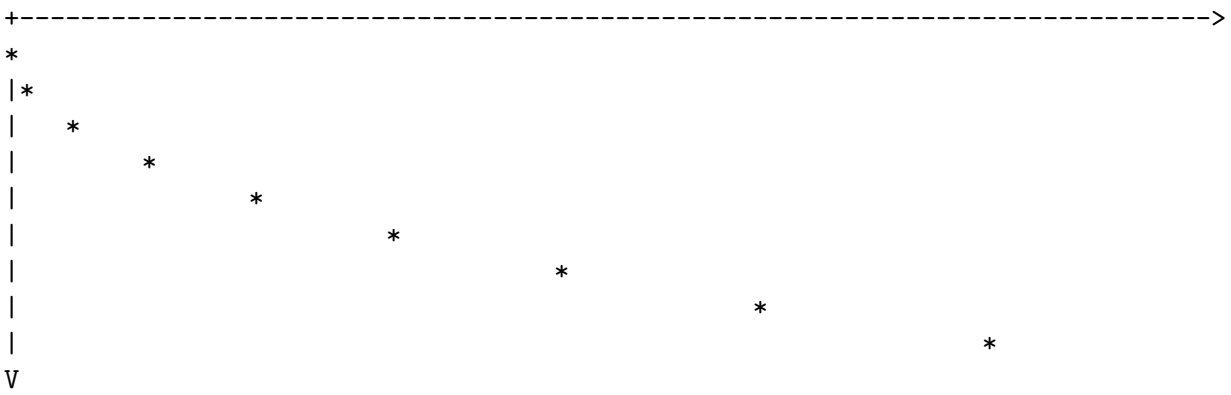

Figura 4: ejemplo de plot ASCII rotado

El programa debe generar en el archivo de salida una gráfica ASCII del vector, pero acostada (es mas sencillo de esta forma).

- En la primera linea se grafica el eje de ordenadas, como un carácter de `'+'` que representa el origen, seguido de 78 guiones y un carácter de `'>'`.
- Por cada elemento del vector se calcula el producto del elemento por el coeficiente y se redondea, llamémosle `valor`, y se imprime una linea de la siguiente forma:
  - Si `valor` es menor que 1 entonces se imprime un `'*'`
  - En caso contrario, se imprime
    - El carácter `'|'`; este sería el eje de las abscisas.
    - Una cantidad de espacios en blanco igual a `(valor - 1)`.
    - El carácter `'*'`.
- Finalmente se imprime una linea con el carácter `'V'`

Por ejemplo si se ejecuta el programa con un archivo de entrada **vector** que contiene el texto `0 1 4 9 16 25 36 49 64` la salida debe ser la que se muestra en al figura **??**

NOTA: *Por simplicidad asumimos que los números son todos positivos. En caso contrario graficamos 0.*

Estamos asumiendo un ancho de pantalla de 80, pero esto bien podría ser un parámetro.

b) (*) Escriba un programa que haga lo mismo que el anterior pero que ahora grafique en vertical.

NOTA: *Observar que para esta parte es necesario recorrer el vector antes de comenzar a imprimir, por lo tanto en este caso la altura máxima se puede conocer. Se sugiere no preocuparse por la eficiencia, por ejemplo se puede leer el archivo de entrada en dos pasadas, la primera para obtener el máximao y el largo y luego obtener la memoria para guardar el vector y volver a leer.*

Nótese que se puede representar la gráfica como una matriz de $m \times n$ caracteres, donde m es el máximo y n es la cantidad de elementos del vector. No es la forma óptima pero es simple para el objetivo de este ejercicio. En el elemento (i, j) de la matriz se guarda un `'*'` o un espacio en blanco. Para imprimir las lineas se sugiere el uso de la función `fwrite` descrita en el apéndice B1.5

*Si se siente con confianza puede intentar graficar también valores negativos. En la figura **??** se muestra un ejemplo de* $\sin(n/10)$*, con n entre 1 y 80 y un coeficiente de 10.*

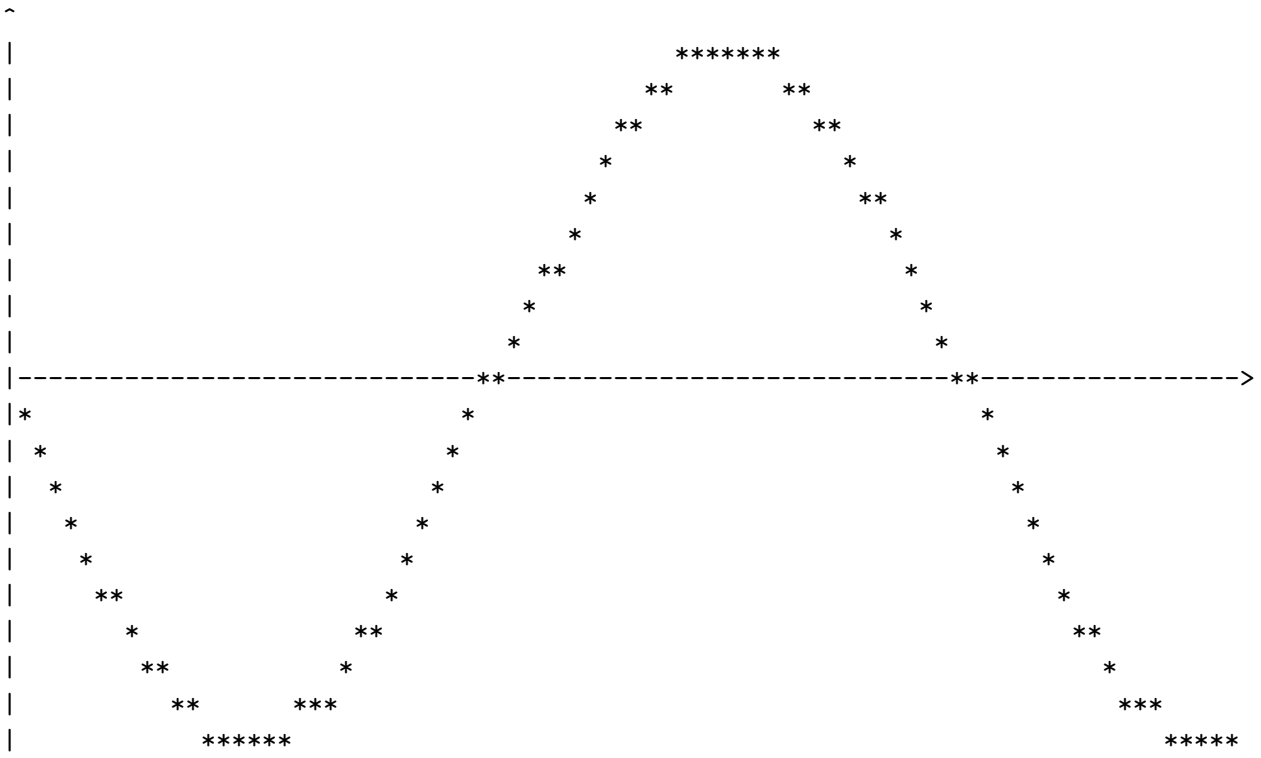

Figura 5: ejemplo de plot ASCII