Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
Computación II (CI-2126)

LABORATORIO 3 (RECURSIÓN Y TDA SECUENCIA)
Posibles soluciones

1.- Construya un programa recursivo que permita calcular el producto de dos números naturales denominados A y B.

Posible solución:

.- La formulación recursiva puede describirse de la siguiente forma:

$$A*B = \begin{cases} A & \text{si B es igual a 1} \\ A*(B-1) + A & \text{si B es diferente a 1} \end{cases}$$

.- He aquí el programa:

/* Muestra como construir una multiplicación como función recursiva */

```
#include <stdio.h>
#include <stdlib.h>

int multiplica (int uno, int dos); /* Prototipo de la función recursiva */

int main ()
{
        int a, b;

        printf ("\nIntroduzca el valor a A: ");
        scanf ("%d", &a);

        printf ("\nIntroduzca el valor a B: ");
        scanf ("%d", &b);

        /* Invocación a la función */
        fprintf (stdout, "\nEl producto de %d por %d es igual a: %d", a, b, multiplica (a,
b));
        fprintf (stdout, "\n");
        exit (0);
}

multiplica (int uno, int dos)
{
        if (dos == 1) return (uno);
        else return(multiplica(uno, dos-1) + uno);
```

```
}
```

2.- Suponga que se define una *h-secuencia* de la siguiente forma:

<h-secuencia> ::= 0 | 1 <h-secuencia> <h-secuencia>

es decir, o es el número cero (0) o es un número uno (1) seguido por dos h-secuencias.

Suponga ahora que de un archivo de texto se suministra una cadena variable de longitud, con caracteres únicamente conformados por los símbolos “1s” y “0s”. Elabore un programa que determine si el vector recibido constituye una h-secuencia.

Posible solución:

Si se estudia la estructura de una h-secuencia se puede establecer las siguientes propiedades:

.- La definición de una h-secuencia puede ser constituida de forma recursiva.
.- Una h-secuencia siempre termina con un cero (0).
.- Se observa que una h-secuencia siempre tiene una longitud impar.
.- Si una h-secuencia tiene una longitud mayor que 1, siempre comienza con un uno (1).
.- En una h-secuencia con longitud mayor que uno (1), la cantidad de ceros (0) es mayor que la de los uno (1).

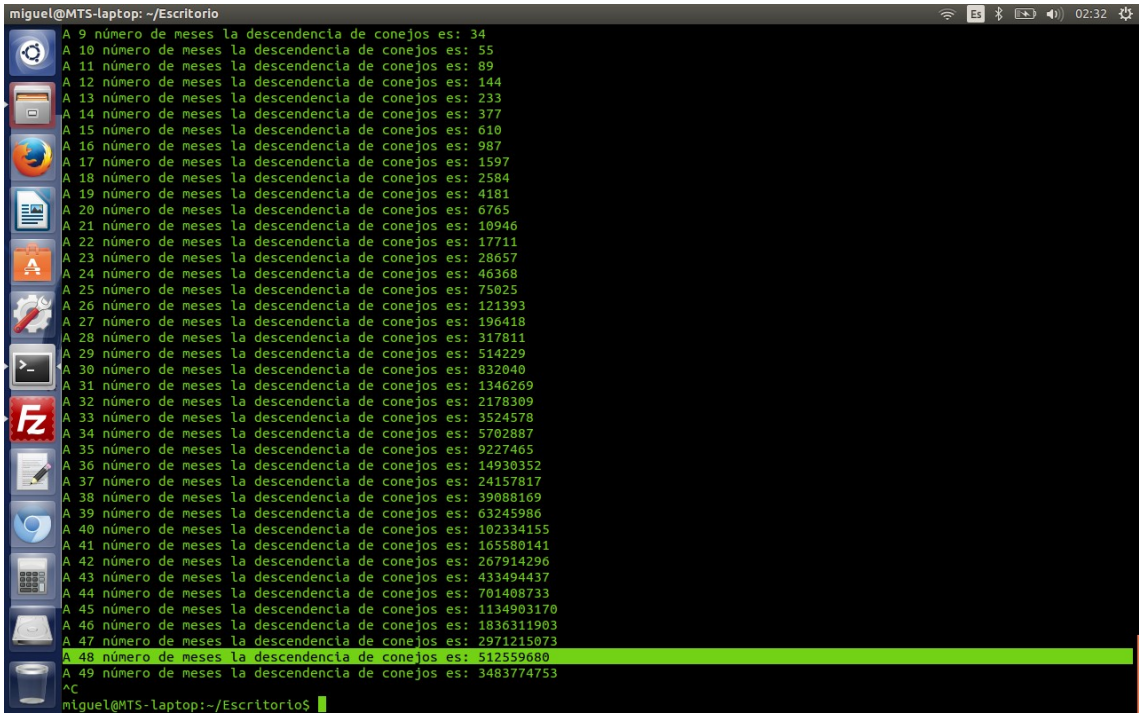
A partir de estas características se puede elaborar el programa.

3.- Ejecute el siguiente programa y responda la pregunta que en el mismo se le plantea.

```
/* Cálculo sorprendente de la serie de Fibonacci v1.1 - MTS                02-02-2011

Versión recursiva del programa que trabaja con la siguiente definición:
F(0)=0
F(1)=1
F(x)=F(x-1) + F(x-2) para x perteneciente a [2,+infinito)
*/

#include <stdio.h>
#include <stdlib.h>
#define LIMITE 55

unsigned int serie_fibonacci (int variable);          /* Prototipo de la impresión del arreglo */

int main ()
{
        unsigned int meses;

        for (meses=1; meses < LIMITE; ++meses)
```

```
                fprintf (stdout, "\nA %d número de meses la descendencia de conejos es: %u", meses, serie_fibonacci (meses));

        exit (0);
}

unsigned int serie_fibonacci (int variable)
{
        switch (variable) {
                case 0: return(0);
                case 1: return(1);
                default: return(serie_fibonacci (variable-1) + serie_fibonacci(variable-2));
        }
}
```

Posible solución:
Una imagen de una posible corrida del programa es:



Como se observa el programa presenta un resultado aparentemente absurdo.

Y aunque está correcto en su lógica, la implementación es fallida ya que olvida que en un computador un entero sin signo tiene un tamaño limitado. Cualquier valor que supere la capacidad de almacenamiento en el mismo, arrojará valores inciertos, ya que el lenguaje C no indicará que hubo un error si no que intentará producir un resultado.

Para el caso en que una arquitectura de computador permite que un entero sin signo ocupe 4 bytes de longitud, el rango de valores posibles es de cero a $2^{32} - 1$. Es decir, de 0

a 4.294.967.295 Cualquier valor que produzca el cálculo de la serie y sea mayor que el límite superior de ese rango, producirá valores que pueden ser erróneos. El término “48” es ese caso, ya que produce un valor que desborda la capacidad de almacenamiento. Hasta el término “47” no se produce ningún error, entonces ya que el resultado de la serie “2.971.215.073” es menor que “4.294.967.295”.

El problema se supera empleando un tipo de dato con mayor capacidad de almacenamiento o generando un TDA propio que integre el manejo de memoria dinámica si se requiere. Hay otras posibilidades como verificar el rango del resultado.

La idea principal del ejercicio es ilustrar a los estudiantes los potenciales problemas que pueden surgir cuando, a pesar de disponer de un algoritmo correcto, la instrumentación es errónea. En este caso es dado que es dependiente de las características propias de un lenguaje y ello puede generar un “bug”. En este caso por desbodarmiento del buffer de representación.

4.- Ejecute varias veces los programas que a continuación se le suministran. Ambos códigos calculan la serie de Fibonacci; uno lo realiza de modo iterativo y otro recursivamente. No emplee valores que generen problemas de desbordamiento. Examine entonces los tiempos de ejecución de ambos códigos utilizando el comando “time” de GNU Linux®.

```
/* Fibonacci - versión iterativa */

#include <stdio.h>

int main()
{
   int x, i;
   unsigned fibn, fibn_1, fibn_2;

   printf("Escriba el numero de meses para los cuales desea calcular: ");
   scanf("%d",&x);

   if (x < 2)
   {
      if (x == 1)
         printf("El numero de Fibonacci cuyo índice es %d es %d\n ", x, 1);
      else
         printf("El numero de Fibonacci cuyo índice es %d es %d\n ", x, 0);
   }
   else
   {
      fibn_1 = 1;
      fibn_2 = 0;
      for(i=2; i <= x; i++)
      {
         fibn = fibn_1 + fibn_2;
```

```
        fibn_2 = fibn_1;
        fibn_1 = fibn;
      }

   printf("El numero de conejos calculados por la serie de Fibonacci cuyos meses son: %d es %u\n", x, fibn);
   }
}
```

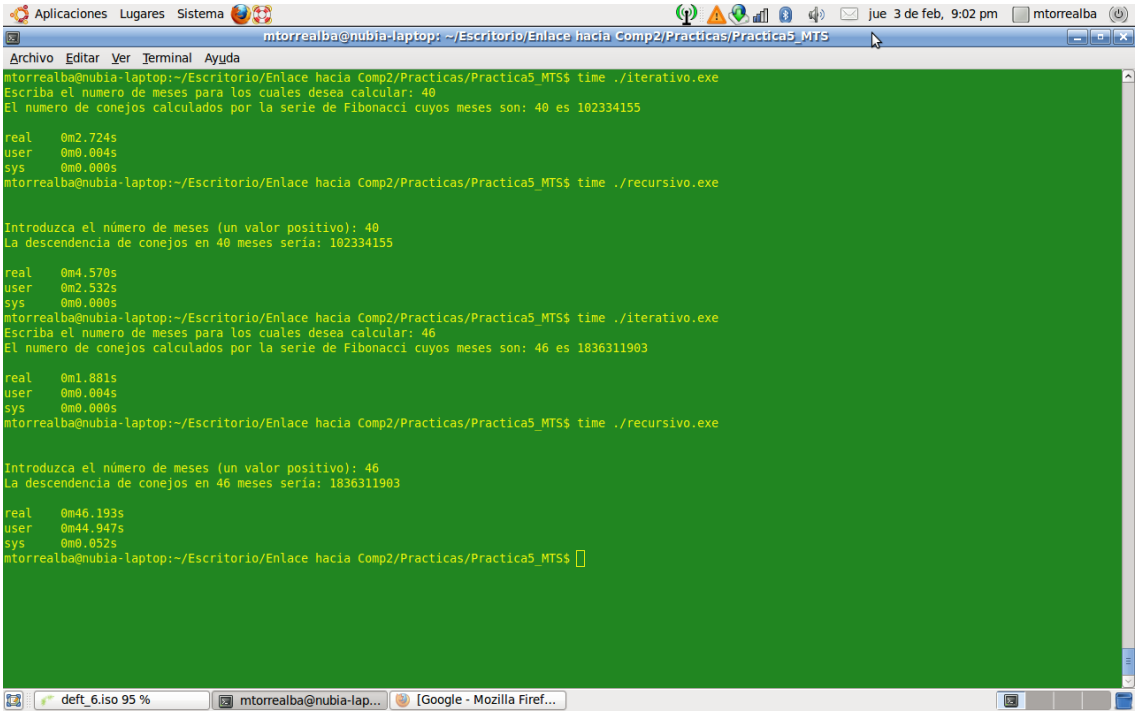
```
/* Cálculo de la serie de Fibonacci v1.1 - MTS                    02-02-2011

Versión recursiva del programa que trabaja con la siguiente definición:
F(0)=0
F(1)=1
F(x)=F(x-1) + F(x-2) para x perteneciente a [2,+infinito)
*/

#include <stdio.h>
#include <stdlib.h>

unsigned int serie_fibonacci (int variable);          /* Prototipo de la impresion del arreglo */

int main ()
{
        unsigned int meses;

        fprintf (stdout, "\n\nIntroduzca el número de meses (un valor positivo): ");
        scanf("%d", &meses);
        fprintf (stdout, "La descendencia de conejos en %d meses sería: %u\n", meses, serie_fibonacci (meses));
        exit (0);
}

unsigned int serie_fibonacci (int variable)
{
        switch (variable) {
                case 0: return(0);
                case 1: return(1);
                default: return(serie_fibonacci (variable-1) + serie_fibonacci(variable-2));
        }
}
```

En caso de que reconozca algún patrón que apunte a que una forma resulta más breve en su ejecución que la otra, explique por qué sucede eso.

Posible solución:

En general, la versión recursiva resulta más lenta que la iterativa. La razón de ello es por la necesidad de apilar datos y ejecutar operaciones para poder retornar apropiadamente a cada invocación. El consumo de recursos genera un "overhead" que se refleja en un aumento del tiempo de ejecución. De trasfondo está implícito el orden de complejidad de ambos programas.

La imagen de abajo muestra dos corridas de ambos programas. Uno se llama "iterativo.exe" y el otro "recursivo.exe"



5.- Suponga que se le suministra la siguiente especificación sintáctica del **TDA Secuencia** de elementos homogéneos provenientes del conjunto E. Para la misma se define una secuencia como una sucesión de elementos y la misma se representa como **S**=<e_o, e_1, e_2, ..., e_{n-1}> con $\mathbf{e_i} \in E \wedge \mathbf{i} \in [0, n-1]$. Para dicha representación se establece que la longitud de S es n, pero el TDA permite la existencia de una secuencia sin elementos, a lo que se denomina *secuencia vacía* y se representa como S={} o < >. Adicionalmente se considera a la operación "~" como la *concatenación* de dos secuencias, hecho que produce como resultado una secuencia. La concatenación puede operar con la secuencia vacía, con una secuencia de un elemento o de *n* elementos. De allí que se puede reescribir la definición de una secuencia como:

$$S = < e_o > \sim < e_1 > \sim < e_2 > \sim \ldots \sim < e_{n-1} > \text{ y la secuencia vacía es: } S: < >$$

En cuanto a la concatenación está queda simbólicamente descrita así:
~: < > x < > ⇢ < > donde < > indica que esa secuencia no contiene ningún elemento
~: < > x < e_0 > ⇢ < e_0 >
~: < e_0 > x < > ⇢ < e_0 >
~: < > x S ⇢ S
~: S x < > ⇢ S

~: S x S´ ⇢ S´´ donde **S** puede ser igual que **S´** o distinto, y **S´´** puede ser igual o distinta de **S**

Por otra parte, con esta definición se derivan las siguientes propiedades:
Propiedad No. 1: < > ~ S = S ~ < > = S
Propiedad No. 2: X ~ (Y ~ Z) = (X ~ Y) ~ Z

De modo que la Especificación Sintáctica queda definida entonces como:
Tipo S [e]

PRIMERO: S ⇢ e donde **S** es no vacía y **e** es el primer elemento de **S**
ÚLTIMO: S ⇢ e donde **S** es no vacía y **e** es el último elemento de **S**
PRINCIPIO: S ⇢ S' donde **S** no es la secuencia vacía y es diferente a **S´**
FINAL: S ⇢ S' donde **S** no es la secuencia vacía y es diferente a **S´**
LONGITUD: S ⇢ n donde **n** es un número natural
VACÍA: S ⇢ < > donde **S** es no vacía al principio y pierde todos sus elementos
INSERTAR: S x i x e ⇢ S' donde **S** es no vacía inicialmente, **i** es la posición en dónde insertar **e**
SUSTITUIR: S x i x e ⇢ S' con **S** es no vacía inicialmente, **i** es la posición dónde reemplazar **e**
BORRAR: S x e ⇢ S' donde **S** no es la secuencia vacía y es diferente a **S´**
ENÉSIMO: S x i ⇢ e donde **i** es la posición de la cuál se requiere el elemento **e**
MAYOR: S x S' ⇢ Lógico donde **S** no es la secuencia vacía y es diferente a **S'** Lógico es verdad si el tamaño de **S** es mayor que el de **S'**
BUSCAR: S x e ⇢ i con **S** no vacía y **e** el elemento del cuál se requiere su primera posición
ORDENAR: S ⇢ S' con **S** no vacía y puede ser diferente a **S'**. Además **S'** queda ordenada
SUBSECUENCIA: S x i x i' ⇢ S' donde **S** es no vacía, **i** es el comienzo de la subsecuencia e **i'** es el final de la subsecuencia

Ejercicio 5.1: Considere que alguien le solicita que desarrolle una representación concreta del objeto abstracto secuencia de elementos. Para esa implementación usted debe emplear una estructura de datos estática que permita registrar un tamaño para cada secuencia. Dicho tamaño estará limitado por el valor **n** por definición. Es decir, $S = < e_o > \sim < e_1 > \sim < e_2 > \sim ... \sim < e_m >$ con $0 < m \leq n-1$

Construya un estructura de datos estática en lenguaje C que satisfaga la consideración descrita y establezca que los elementos de la secuencia son valores enteros sin signo.

Ejercicio 5.2: Codifique en lenguaje C la implementación de las rutinas:
INSERTAR
ORDENAR

Este ejercicio solamente se planteará mañana. Se terminará la discusión el siguiente lunes. Después que ellos traigan sus propuestas.

Para mañana se aclarará el mismo y de ser necesario se tratará la especificación semántica -sin incluir complejidad algorítmica- Ver el documento adjunto.

19 de Septiembre de 2014

MTS/mts