INSTITUTO POLITÉNICO NACIONAL
ESCUELA SUPERIOR DE CÓMPUTO

ESTRUCTURAS DE DATOS

PRÁCTICA #4

SOLUCIONES RECURSIVAS

GRUPO: 1CM8

EQUIPO: LAS MÁS PERRAS
INTEGRANTES:

- JIMÉNEZ DELGADO LUIS DIEGO 2019630461
- SÁNCHEZ CASTRO AARÓN GAMALIEL 2019630079
- SÁNCHEZ TIRADO CITLALI YASMÍN 2019630096

PROFESOR: EDGARDO ADRIÁN FRANCO MARTÍNEZ
FECHA DE ENTREGA: 13 DE MAYO 2019

INTRODUCCIÓN

Mediante la realización de estas prácticas nos encontramos con conceptos de recursividad, sus funciones, sus ventajas y sus desventajas, ya que al realizar las prácticas nos dábamos cuenta en qué tipo de problemas era más eficiente el hecho de utilizar una función recursiva. Así mismo sabemos que el concepto de recursividad puede ser un poco difícil de entender al principio pero con práctica se puede aclarar este.

MARCO TEÓRICO

RECURSIVIDAD

La recursividad (recursión) es aquella propiedad que posee una función por la cual dicha función puede llamarse a sí misma. Se puede utilizar la recursividad como una alternativa a la iteración. Una solución recursiva es normalmente menos eficiente en términos de tiempo de computadora que una solución iterativa debido a las operaciones auxiliares que llevan consigo las llamadas suplementarias a las funciones; sin embargo, en muchas circunstancias el uso de la recursión permite a los programadores especificar soluciones naturales, sencilla, en caso contrario, difíciles de resolver. Por esta causa, la recursión es una herramienta poderosa e importante en la resolución de problemas y en la programación.

La naturaleza recursiva

En algunos problemas es útil disponer de funciones que se llamen a sí mismo ya sea directa o indirectamente. En matemáticas existen numerosas funciones que tienen carácter recursivo; de igual modo numerosas circunstancias y situaciones de la vida ordinaria tienen carácter recursivo.

Recursividad directa e indirecta

Cuando un procedimiento incluye una llamada a sí mismo se conoce como recursión directa.
Cuando un procedimiento llama a otro procedimiento y éste causa que el procedimiento original sea invocado, se conoce como recursión indirecta.

Un requisito para que un algoritmo recursivo sea correcto es que no genere una secuencia infinita de llamadas sobre sí mismo. Cualquier algoritmo que genere una secuencia de este tipo no puede terminar nunca. En consecuencia la definición recursiva debe incluir una condición de salida, que se denomina componente base, en el que f(n) se defina directamente (es decir, no recursivamente) para uno o más valores de n.
En definitiva, debe existir una “forma de salir” de la secuencia de llamadas recursivas.

Caso recursivo: una solución que involucra volver a utilizar la función original, con parámetros que se acercan más al caso base. Los pasos que sigue el caso recursivo son los siguientes:

1
- El pricedimiento se llama así mismo.

2
- El problema se resuelve, tratando el mismo problema pero de tamaño menor

3
- La manera en la cual el tamaño del problema disminuye asegura que el caso base eventualmente se alcanzará

¿Por qué escribir programas recursivos?

- Son más cercanos a la descripción matemática.
- Generalmente más fáciles de analizar
- Se adaptan mejor a las estructuras de datos recursivas.
- Los algoritmos recursivos ofrecen soluciones estructuradas, modulares y elegantemente simples.

DISEÑO Y FUNCIONAMIENTO DE LA SOLUCIÓN

IMPLEMENTACIÓN DE LA SOLUCIÓN

Planteamiento del problema

Programar en ANSI C la implementación recursiva del término n de la serie Fibonacci. En matemáticas, la serie de Fibonacci es la siguiente sucesión infinita de números naturales:

$$0,1,1,2,3,5,8,13,21,34,55,89,144\ldots$$

El primer elemento es 0, el segundo es 1 y cada elemento restante es la suma de los dos anteriores.

Implementación de la solución

IMPLEMENTACIÓN PARA LA SERIE FIBONACCI

```
/*
    FUNCIÓN: fibonacci (int n,int m, int limit,int a)
    RECIBE: int n (PRIMER NÚMERO A SUMAR), int m (SEGUNDO NÚMERO A SUMAR),
        int limit (POSICIÓN LÍMITE), int a (POSICIÓN ACTUAL).
    DEVUELVE: SE LLAMA A SÍ MISMA SI ES NECESARIO.
    DESCRIPCIÓN: FUNCIÓN ENCARGADA DE CALCULAR LOS NÚMEROS DE LA SERIE DE
FIBONACCI.
    OBSERVACIONES: CUANDO EL NÚMERO ES MUY GRANDE IMPRIME NEGATIVOS.
*/
void fibonacci(int n,int m,int limit,int a){
if(n==0&&m==1){
printf("0, 1, ");
}
if(a==limit){
return;
}
printf("%i, ",m+n);
fibonacci(m,m+n,limit,a+1);
}
```

Función fibonacci encargada de imprimir los números de la serie, se llama con una nueva posición para calcular uno nuevo.

```
int main(){
int n;
printf("Ingresa el numero limite:\n>");
scanf("%i",&n);
fibonacci(0,1,n,1);
return0;
}
```

IMPLEMENTACIÓN PARA LA SERIE TRIBONACCI

```
/*
    FUNCIÓN: tribonacci (int x,int y, int z, int limit)
    RECIBE: int x (PRIMER NÚMERO A SUMAR), int y (SEGUNDO NÚMERO A SUMAR),
        int z (TERCER NÚMERO A SUMAR), int limit (POSICIÓN LÍMITE), int n
        (POSICIÓN ACTUAL).
    DEVUELVE: SE LLAMA A SÍ MISMA SI ES NECESARIO.
    DESCRIPCIÓN: FUNCIÓN ENCARGADA DE CALCULAR LOS NÚMEROS DE LA SERIE DE
TRIBONACCI.
    OBSERVACIONES: CUANDO EL NÚMERO ES MUY GRANDE IMPRIME NEGATIVOS.
*/
void tribonacci(int x,int y,int z,int limit,int n){
if(x==0&&y==0&&z==1){
printf("0, 0, 1, ");
```

```
}
if(n>=limit){
return;
}
printf("%i, ",x+y+z);
tribonacci(y,z,x+y+z,limit,n+1);
}
```

Función tribonacci encargada de imprimir los números de la serie, se llama con una nueva posición para calcular uno nuevo.

```
int main(){
int n;
printf("Ingresa el nÃ°mero limite:\n>");
scanf("%i",&n);
tribonacci(0,0,1,n,1);
return0;
}
```

IMPLEMENTACIÓN PARA LA SERIE TRIBONACCI

```
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#include <time.h>
#include <iostream>
//usging namespace std;

    /*
    Función: imprime
    Precondición:
                    *tab    Puntero a una matriz de tipo entero.
                    fil     Entero que indica el numero de filas de la matriz.
                    col     Entero que indica el numero de columnas de la matriz.
                    disc    Parámetro de tipo entero que indica el numero de discos
usados.
                    ultNum  Entero que indica el numero que esta usando el disco
mas grande.
*/
Esta function es la encargada de imprimir los discos
void imprime( int *tab, int fil, int col, int ultNum )
{



    int f, c;
    int i, esp;


    for( c=col-1; c >= 0; c-- )
    {
        for( f=0; f < fil; f++ )
        {
            esp = ( ultNum - tab[col*f+c] )/2;

            // Espacios a la izquierda
            for( i=0; i < esp; i++ )
                printf( " " );

            // Imprime los comodines
            for( i=0; i < tab[col*f+c]; i++ )
                printf("%c",220);

            // Espacios a la derecha
            for( i=0; i < esp; i++ )
                printf( " " );

            printf( "\t" );
        };

        printf( "\n" );
    };


};

    /*
    Función mueveDisco
    Precondición:
                    *tab    Puntero a una matriz de tipo entero.
```

```
                    fil     Entero que indica el numero de filas de la matriz.
                    col     Entero que indica el numero de columnas de la matriz.
                    disc    Parámetro de tipo entero que indica el numero de discos
usados.
                    ultNum  Entero que indica el numero que esta usando el disco
mas grande.
                    filOrig Entero que indica el numero de fila de la matriz en la
cual hay que agarrar el disco
                    filDest Entero que indica el numero de fila de la matriz en la
cual hay que dejar el disco.
    Poscondición:
                    Se mueve el disco y se llama a la función que imprime el
tablero.
    */
```

Esta función es la encargada de mover los discos a las columnas correspondientes.

```
void mueveDisco( int *tab, int fil, int col, int ultNum, int filOrig, int filDest )
{

    int cO=col-1, cD=col-1;


    // Se busca el disco que se encuentre mas arriba y por lo tanto el mas pequeño
de la fila de origen.
    while( cO >= 0  &&  tab[col*filOrig+cO] == 0 )
    {
        cO--;
    };
    if( cO < 0 )
        cO = 0;

    // Ahora se calcula cual es la posición libre mas arriba de la fila de destino
    while( cD >= 0  &&  tab[col*filDest+cD] == 0 )
    {
        cD--;
    };

    // Se mueve el disco de la fila de origen a la de destino:
    tab[col*filDest+cD+1] = tab[col*filOrig+cO];
    tab[col*filOrig+cO] = 0;

    // Se imprime el tablero:
    imprime( tab, fil, col, ultNum );
};

/*
Función: hanoi
Precondición:
                *tab    Puntero a una matriz de tipo entero.
                fil     Entero que indica el numero de filas de la matriz.
                col     Entero que indica el numero de columnas de la matriz.
                disc    Parámetro de tipo entero que indica el numero de discos
usados.
                ultNum  Entero que indica el numero que esta usando el disco mas
grande.
                O,A,D   Tres enteros que indican la fila desde donde se toma  el
disco y a donde se debe mover. La primera vez que se llama a hanoi tienen los
valores de: 0 ,1 y 2 respectivamente.
Poscondición:
                Se llama recursivamente a hanoi hasta resolver el tablero.
*/
```

Aquí hacemos los calculus necesarios para mover los discos asi como el tiempo, ya que tenemos una variación dependiendo cuantos discos sean.

```
void hanoi( int *tab, int fil, int col, int disc, int ultNum, int O, int A, int D )
{

    if( disc==1 )
    {
        // Se borra la pantalla, se imprime la tabla
        system("cls");
        mueveDisco( tab, fil, col, ultNum, O, D );
        //system("sleep ns") solo funciona en linux
        //if(col<=5) system("sleep 0.8"); else if(col<=10) system("sleep 0.3");
else if(col<=15) system("sleep 0.06"); else if(col>15) system("sleep 0.02");
        if(col<=5) Sleep(800); else if(col<=10) Sleep (300); else if(col<=15)
Sleep(60); else if(col>15) Sleep(20);
    }
    else
    {
```

```
        hanoi( tab, fil, col, disc-1, ultNum, O, D, A );

        system("cls");
        mueveDisco( tab, fil, col, ultNum, O, D );
        //system("sleep ns") solo funciona en linux
        //if(col<=5) system("sleep 0.8"); else if(col<=10) system("sleep 0.3");
else if(col<=15) system("sleep 0.06"); else if(col>15) system("sleep 0.02");
        if(col<=5) Sleep(800); else if(col<=10) Sleep (300); else if(col<=15)
Sleep(60); else if(col>15) Sleep(20);
        hanoi( tab, fil, col, disc-1, ultNum, A, O, D );
    };

};
```

Esta es nuestra función principal encargada de pedir el numero de discos y así mismo darnos el espacio o bien el tablero.

```
main()
{
    int fil=3, col, *tablero = NULL;
    int f, c, disc=1, ultNum;


    printf( "Introduce el numero de discos : " );
    scanf( "%i", &col );


    tablero = (int *)malloc( sizeof(int)*fil*col );


    // borra las torres poniendo "los discos" en una de ellas y 0 en el resto.
    for( f=0; f < fil; f++ )
        for( c=col-1; c >= 0; c-- )
            if( f==0 )
            {
                tablero[col*f+c] = disc;
                disc+=2;
            }
            else
                tablero[col*f+c] = 0;

    ultNum = disc;

    // Se imprime el tablero antes de iniciar ningún movimiento
    system("cls");
    imprime( tablero, fil, col, ultNum );
    Sleep(1000);


    // Se llama a hanoi para comenzar
    hanoi( tablero, fil, col, col, ultNum, 0, 1, 2 );
};
```

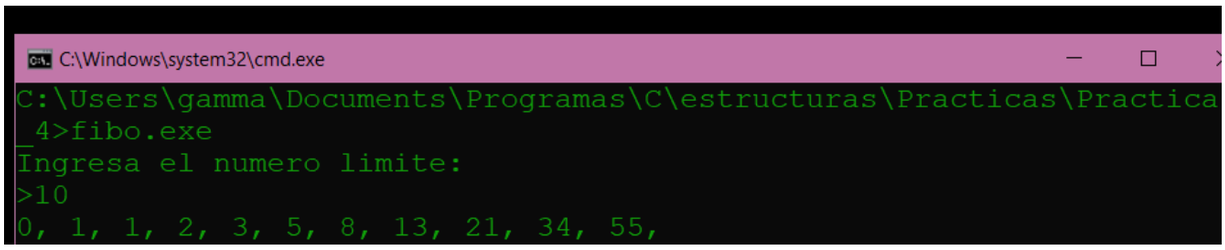
IMPLEMENTACIÓN PARA MERGESORT

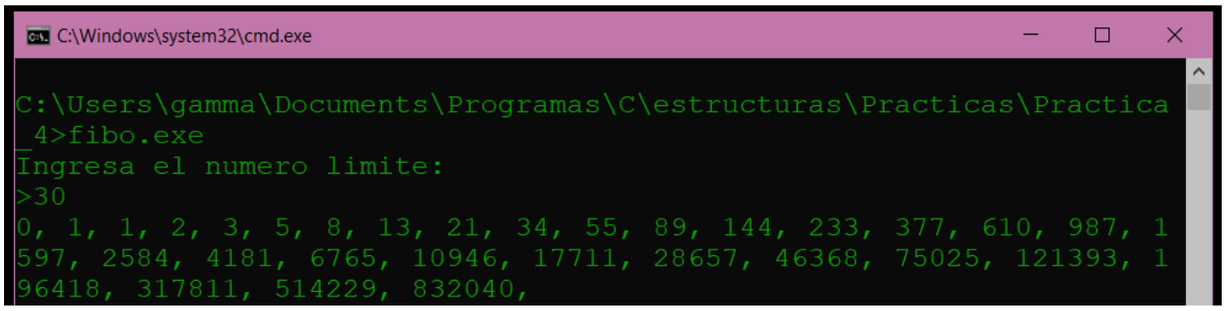
```
#include <time.h>
#include <stdlib.h>
#include <stdio.h>

int *cal(int arreglo[], int tamano);
void printArray(int arrTo[], int tam);
void copyArray(int *arrTo, int *arrFrom, int tam);

int main()
{
    int tam = 26, i;
    int arreglo[tam];
    srand(time(NULL));
    for (i = 0; i < tam; ++i)
    {
        if (i % 2 == 0){
            arreglo[i] = i;
        }
        else
        {
            arreglo[i] = -i + 2;
        }
    }
    printf("\a\n");
    printArray(arreglo, tam);

    copyArray(arreglo, cal(arreglo, tam), tam);
    printf("\resultado\n");
    printArray(arreglo, tam);
```

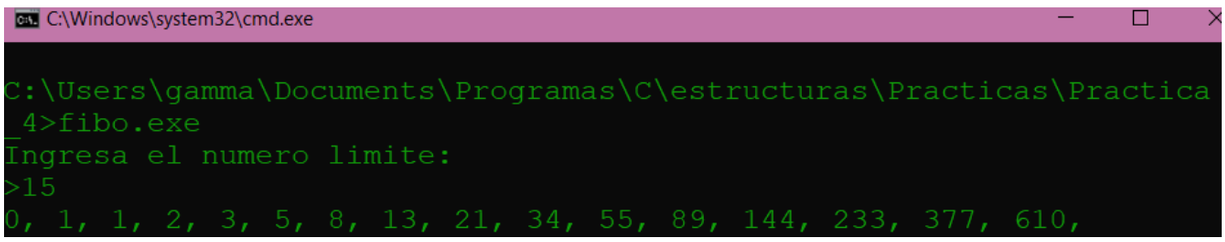
```
    return 0;
}

/*
    FUNCIÓN: printArray(int arrTo[], int tam)
    RECIBE: el arreglo de int a imprimir int arrTo[], el tamaño del arreglo int tam
    DEVUELVE: void;
    DESCRIPCIÓN:
    Funcion que recibe un arreglo de tamaño n, e imprime todos los valores en el
separados por un /
    OBSERVACIONES:
*/
void printArray(int arrTo[], int tam)
{
    int i = 0;
    for (i = 0; i < tam; ++i)
    {
        printf("%i \t", arrTo[i]);
    }
    printf("\n");
}

/*
    FUNCIÓN: copyArray(int *arrTo, int *arrFrom, int tam)
    RECIBE: el arreglo al que se le van asignar los numeros int *arrTo,
     el arreglo del que se  van asignar los numeros int *arrFrom,
        el tamaño del arreglo  int tam
    DEVUELVE: void
    DESCRIPCIÓN: copia todos los elementos de un arreglo a otro
*/
void copyArray(int *arrTo, int *arrFrom, int tam)
{
    int i = 0;
    for (i = 0; i < tam; ++i)
    {
        arrTo[i] = arrFrom[i];
    }
}

/*
    FUNCIÓN: *cal(int arreglo[], int tamano)
    RECIBE:  el arreglo a acmodar int arreglo[], el tamaño del arreglo  int tamano
    DEVUELVE: un arreglo de tamño variable int *
    DESCRIPCIÓN: Realiza el proceso de ordenamiento merge por medio de recursividad
*/
int *cal(int arreglo[], int tamano)
{
    printf("\n");
    if (tamano != 1)
    {
        int i, j;
        int tamArr = tamano / 2;
        int tamArr2 = tamArr + (tamano % 2);
        int arr1[tamArr];
        int arr2[tamArr2];
        for (i = 0; i < (tamArr); ++i)
        {
            arr1[i] = arreglo[i];
            arr2[i] = arreglo[tamArr + i];
        }
        arr2[tamArr2 - 1] = arreglo[tamano - 1];
        copyArray(&arr1, cal(arr1, tamArr), tamArr);
        copyArray(&arr2, cal(arr2, tamArr2), tamArr2);
        int res[tamano], k = 0;
        i = 0;
        j = 0;
        while (i <= tamArr && j <= tamArr2 && k < tamano)
        {
            if (i <= tamArr && arr1[i] < arr2[j])
            {
                res[k] = arr1[i];
                ++i;
            }
            else
            {
                if (j < tamArr2)
                {
                    res[k] = arr2[j];
                    ++j;
                }
                else
                {
                    res[k] = arr1[i];
                    ++i;
                }
            }
            ++k;
        }
        return res;
    }
```

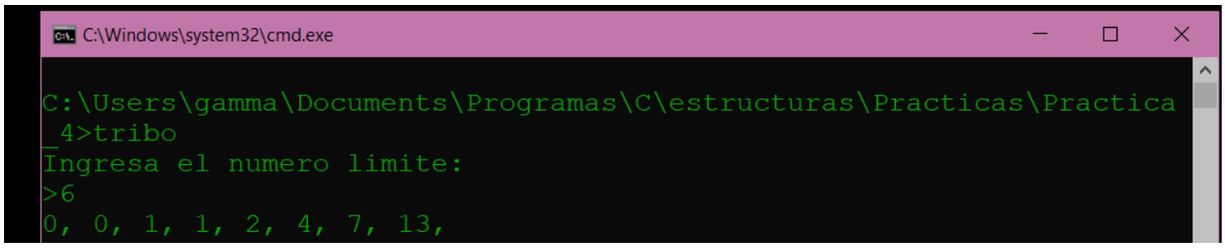
```
    return arreglo;
}
```

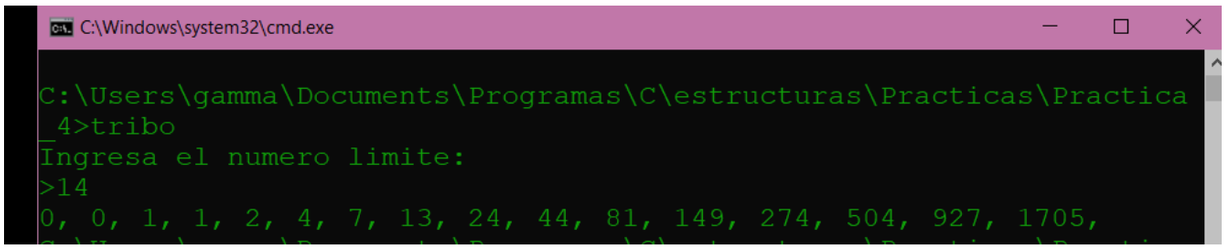
Funcionamiento

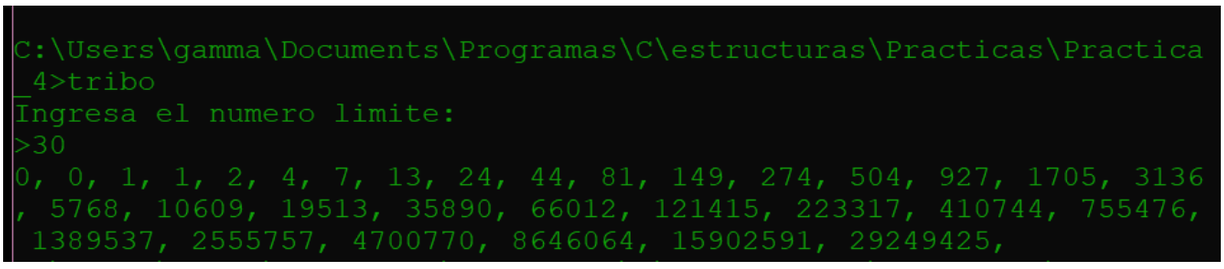




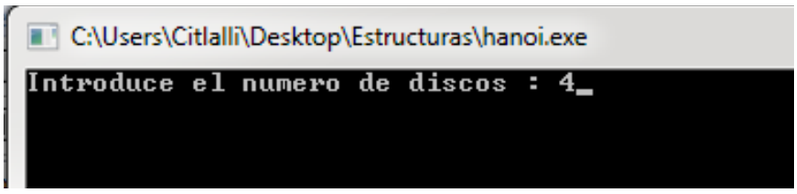


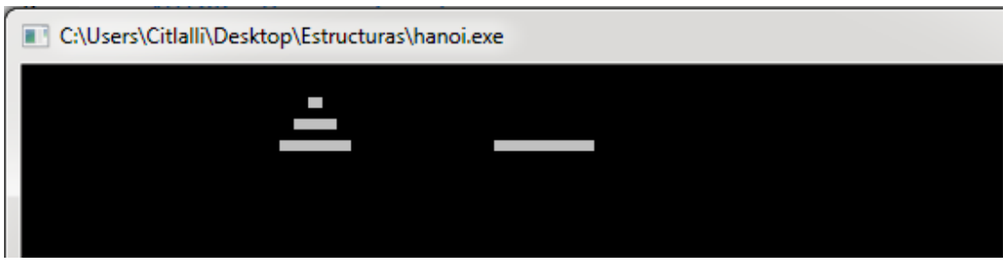




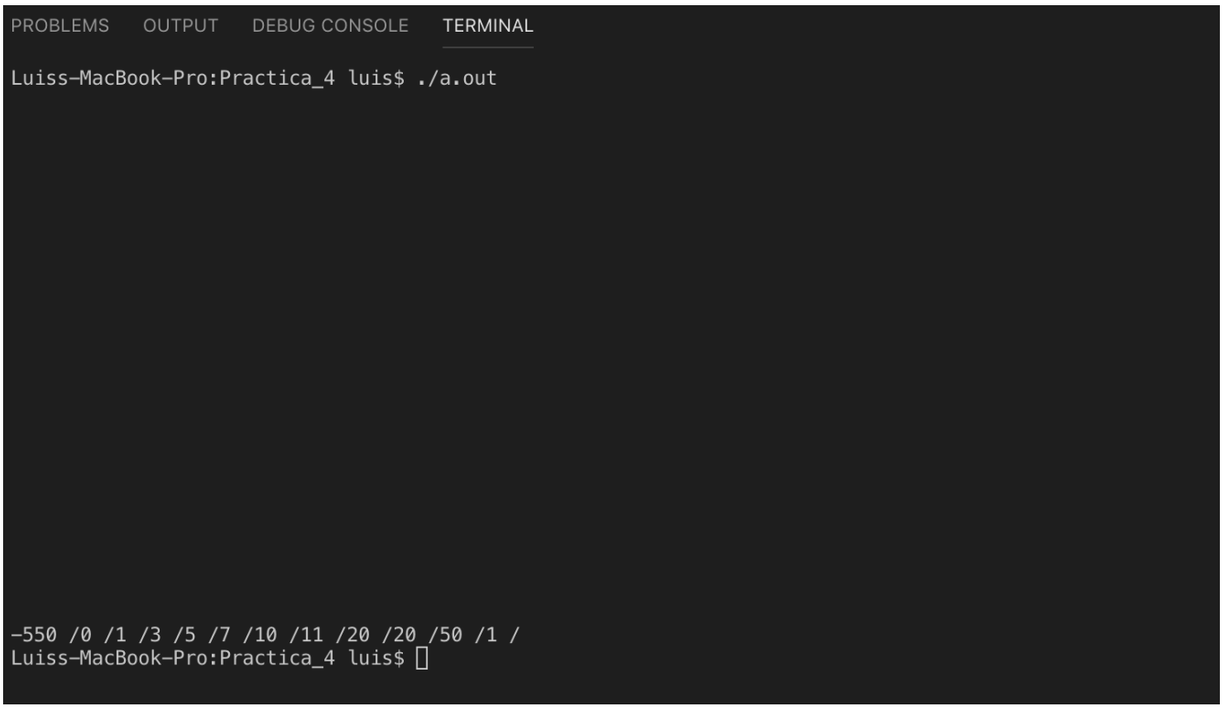


Funcionamiento de torres de Hanoi





Funcionamiento de MERGE SORT



ERRORES DETECTADOS

Ambos programas al calcular números grandes imprimen números negativos. El de merge después de usar

POSIBLES MEJORAS

Soporte para números grandes por parte de ambos programas. Para el de merge se requiere utilizar de listas para la solución de este problema del desbordamiento de memoria.

CONCLUSIONES

Jiménez Delgado Luis Diego:
Los mejroes ejemplos de mejoramiento de procesos por medio de la recursividad, así como la cantidad de procesos que realiza el merge sort, mejora de manera considerable como es que razona uno para el mejoramiento de procesos y solución de algoritmos en la computadora. Por así decirlo, facilita de manera inmediata el uso de algoritmos a nivel computational y su solución sencilla.

Sánchez Castro Aarón Gamaliel:

Al realizar esta práctica es posible mejorar la hablidad de abstraer procesos para ejecutarlos de manera recursiva, lo anterior provoca que los códigos implementados en la práctica sean bastante sencillos y fáciles de entender si se tiene idea de qué es la recursión. Cabe mencionar que implementar una solución recursiva no siempre es mejor, existen casos en los cuales gaste más recursos; por lo tanto, es indispensable analizar si la solución que proponemos para un problema es posible adaptarla a la recursividad y si realmente es conveniente.

Sanchez Tirado Citlalli Yasmin:
En el desarrollo de esta práctica logro tener una idea más clara de lo que es la recursividad, se puede definir como la propiedad de llamarse así misma el número de veces necesarias hasta lograr llegar al caso base. La recursión al igual que la iteración pueden tornarse infinitas y terminar debido al agotamiento de memoria de la computadora, pero por obviedad esto no es recomendable.
Se puede decir que la recursividad es una técnica de programación bastante útil y muy interesante de estudiar.
Una de las desventajas de la recursión es que invoca el mecanismo en forma repetida y, por lo tanto, sobrecarga las llamadas de función. Esto puede resultar costoso tanto en tiempo de procesador como en espacio de memoria.