Recursión

Objetivos

- Practicar el diseño de algoritmos recursivos, comprender su ejecución, así como ventajas y desventajas de la recursividad.
- Mostrar la recursividad como herramienta cuando realmente es necesaria y también como técnica para simplificar o dividir un problema en problemas de menor tamaño del mismo tipo.

En este práctico se asume definido el tipo para enteros sin signo (naturales):

```
typedef unsigned int uint;
```

Ejercicio 1 Sucesión de Fibonacci

Implemente de forma recursiva la función `fib` que permite calcular los valores de la sucesión de Fibonacci $(0, 1, 1, 2, 3, 5, 8, 13, 21....)$. Esta sucesión se representa mediante la fórmula $F_n = F_{n-1} + F_{n-2}$. Es decir, cada elemento es la suma de los dos anteriores, siendo el primer elemento $F_0 = 0$, y el segundo $F_1 = 1$. Por lo tanto $F_2 = 1 + 0,\ F_3 = 1 + 1,\ F_4 = 2 + 1,\ F_5 = 3 + 2$ y así sucesivamente.

Ejercicio 2 Es Palindrome

Implemente un algoritmo recursivo que determina si un string es palíndrome. El string está representado en un arreglo.

Ejercicio 3 Rayuela

En una hilera de una calle con adoquines unos niños juegan a la rayuela. Para esto numeran los adoquines como se indica en la Figura 1. Los movimientos permitidos del juego son:

- Al principio del juego los niños se ubican en el adoquín 0.
- De un adoquín numerado i se puede saltar al adoquín numerado $i + 1$.
- De un adoquín numerado i se puede saltar al adoquín numerado $i + 2$ (sin pasar por el adoquín $i + 1$).

Implemente un algoritmo recursivo que calcule **el número de caminos posibles** para alcanzar un adoquín objetivo numerado con *n* (mayor que cero). Asuma que la cantidad de caminos para llegar al adoquin 0 es 1. Por ejemplo, el número de caminos posibles para *n*=3 es 3 y son los siguientes: (0,1,2,3), (0,2,3) y (0,1,3).

Figura 1: Esquema de adoquines

Calcule la cantidad de caminos para n desde 1 hasta 10.

Ejercicio 4 Insertion Sort

(a) Implemente de forma recursiva la función `ordenar`:

```
/* Ordena A[1..n] de manera creciente. */
void ordenar(float * A, uint n);
```

utilizando la función auxiliar `insertarOrdenado`:

```
/* Inserta e en A[1..n+1] de manera ordenada.
   Precondición: n>=0. Si n>=1 => A[1..n] está ordenado de manera ↩
      creciente. A[n+1] es indeterminado.
   Postcondición: A[1..n+1] queda ordenado de manera creciente. */
void insertarOrdenado(float * A, uint n, float e);
```

(b) Implemente la función auxiliar `insertarOrdenado` utilizando recursión.

(c) Demuestre por inducción que el algortimo de la función auxiliar `insertarOrdenado` es correcto.

(d) Transforme el algoritmo recursivo de la función auxiliar `insertarOrdenado` en iterativo.

Ejercicio 5 Máximo Común Divisor

Implemente un algoritmo recursivo que calcule el máximo común divisor de dos enteros positivos.

Ejercicio 6 Torres de Hanoi

Escenario: existen tres cilindros verticales, A, B y C, en los que es posible insertar discos. En el cilindro A hay n discos todos de diferente tamaño, colocados en orden de tamaño con el más chico arriba. Los otros dos cilindros están vacíos. El problema es **pasar la torre de discos al cilindro C** usando como único movimiento elemental el cambio de un disco de un cilindro a otro cualquiera, **sin que en ningún momento un disco vaya a colocarse encima de otro más chico que él.** La Figura 2 presenta los estados inicial y final de este problema para 3 discos.

Implememte un algoritmo para **imprimir las secuencias de movimientos elementales** que resuelvan el problema de las torres de Hanoi, de cualquier tamaño. Los movimientos elementales son de la forma: `MOVER DISCO i DEL cilindro x AL cilindro y`.

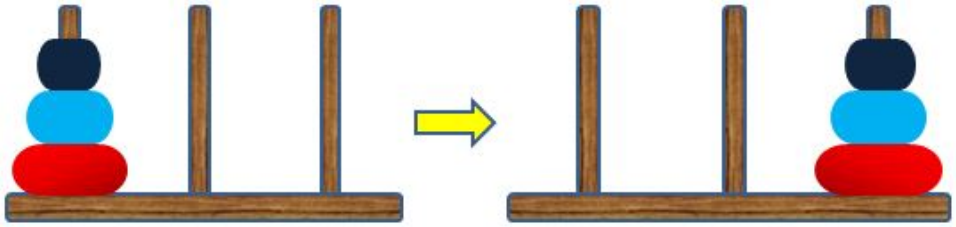

Figura 2: Estado inicial y final para el problema de las Torres de Hanoi con *N*=3[1]

Compruebe que la cantidad de movimientos que se deben realizar es $2^n - 1$.

[1] Fuente: `http://deevybee.blogspot.com.ar/2013/04/forget-tower-of-hanoi-new-ecologically.html`

Ejercicio 7 Factorial

La versión recursiva vista de `factorial` no presenta recursión de cola. Use un acumulador para obtener una versión recursiva que presente recursión de cola.

```
/* Devuelve el producto de factorial(n) por 'acum'. */
uint factAcum(uint n, int acum);
```

Ejercicio 8 Evaluación de Polinomio

Implemente un algoritmo recursivo que evalúe un polinomio usando la regla de Horner:

$$P(x) = a_0 + x(a_1 + x(a_2 + \cdots + (xa_n)\cdots))).$$

```
/* Evalúa P(x):
   P[0] + ... + P[i] x^i + ... + P[n] x^n. */
float horner(float * P, uint n, float x);
```

Ejercicio 9 Combinaciones

Implemente un algoritmo recursivo que calcule las combinaciones de m tomadas de a n, donde $m \geq n$. Asuma las siguientes ecuaciones:

$$\begin{aligned} C(m,m) &= 1 \\ C(m,0) &= 1 \\ C(m,1) &= m \\ C(m,n) &= C(m-1,n) + C(m-1,n-1), \text{ si } m > n > 0 \end{aligned}$$

NOTA: la expresión utilizada en el paso recursivo se conoce como *Identidad de Pascal*.

Ejercicio 10 Contar unos en matriz

(a) Implemente la función `contarUnosVector` que determine la cantidad de celdas de valor 1 en un vector v de largo m. El vector está definido como un puntero a un arreglo de m entradas. La función debe ser implementada de manera recursiva.

```
/* Cuenta los 1s del vector*/
int contarUnosVector(int* v, uint m);
```

(b) Implemente la función `contarUnosMatriz` que determine la cantidad de celdas valor 1 en una matriz A. La matriz está definida como un puntero a un arreglo de mxn entradas donde m es la cantidad de filas y n la de columas. La función debe ser implementada de manera recursiva. Utilice como caso base la función de la parte (a), es decir descomponga la matriz eliminando filas hasta llegar al caso de que la matriz es de 1xn.

Nota: Tenga en cuenta que para acceder a la posición A(i,j) representada como un arreglo debe acceder a la posición del arreglo A[i*m+j].

```
/* Cuenta los 1s de la matriz */
int contarUnosMatriz(int * matriz, uint cantFilas, uint m);
```

(c) Ahora asuma que la matriz M es cuadrada y su dimensión es 2^n x 2^n (donde n es un número natural mayor o igual que 0) ¿Puede hacerse una recursión más inteligente? Haga una implementación recursiva que aproveche esta característica de la matriz.

Ejercicio 11 Vasos alternados

Se tiene una disposición de n vasos, donde n es par. Los vasos de la primera mitad están llenos y los de la segunda mitad están vacíos. Se quiere dejar los vasos alternados: los vasos de las posiciones impares deben quedar llenos y los de las posiciones pares deben quedar vacíos. Sólo se permite verter el contenido de vasos en otros (es decir, no se puede desplazar vasos).

Implemente un algoritmo recursivo que resuelva el problema. En la siguiente especificación el valor `true` representa un vaso lleno y el valor `false` un vaso vacío.

```
  /* Modifica V. Solo se permite intercambio entre posiciones.
   Precondición: n % 2 == 0
                 V[1..n/2] = {true}, V[n/2 + 1.. n] = {false}.
   Postcondición: V[i] = true si y solo si i%2 == 1. */
void vasos(bool * V, uint n);
```

Ejercicios complementarios

Ejercicio 12 Invocaciones Fibonacci

(a) Implemente un algoritmo recursivo que calcule la cantidad de invocaciones que deben hacerse para obtener Fibonacci(n), el $n-$ésimo número de la secuencia de Fibonacci, con el algoritmo visto en el curso. Asuma que los primeros dos números de la secuencia son 1 y 1.

Calcule este valor para los primeros 10 números. ¿Cómo se relacionan con los números de Fibonacci?

(b) Una secuencia $a_1, a_2, a_3, \cdots, a_i, \cdots$ se dice que es aditiva si $a_i = a_{i-2} + a_{i-1}$ para $i \geq 3$. Implemente un algoritmo recursivo que calcule el n-ésimo término de una secuencia aditiva y solo requiera n invocaciones para lograrlo.

```
/* Devuelve el n-ésimo término de la secuencia aditiva que empieza↩
    con a1 y a2. */
int secAditiva(uint n, int a1, int a2);
```

Implemente una versión recursiva de Fibonacci que solo realice n invocaciones para calcular el n-ésimo término.

(c) Implemente un algoritmo que devuleve el n-ésimo y el $n-1$-ésimo número de Fibonacci.

```
struct par {
  int primero, segundo;
};

/* Devuelve (Fibonacci(n-1), Fibonacci(n)).
  Precondición: n > 1. */
par fibAnterior(uint n);
```

Ejercicio 13 Función Exponencial

La función exponencial (e^x) se define mediante:

$$\exp(x) = 1 + x + x^2/2 + x^3/6 + \cdots x^i/i! + \cdots .$$

En particular para $0 \leq x < 1$ cada término es menor al anterior. Implemente un algoritmo recursivo que obtenga una aproximación del valor e^x truncando el cálculo al llegar a un término menor a una tolerancia establecida. Además se debe devolver cuántos términos fueron necesarios para obtener el resultado.

```
/* Devuelve la aproximación de e^x hasta el término que sea menor a 'tol'.
   Devuelve en 'cantidad' la cantidad de términos evaluados. */
float exp(float x, float tol, uint & cantidad);
```

Ejercicio 14 Regla graduada

Se considera una regla graduada en la que la longitud de las marcas indica la importancia, esto es, a las posiciones enteras corresponde longitud máxima, p, a las posiciones de la forma x,5 corresponde longitud $p-1$, a las posiciones de la forma x,25 o x,75 corresponde longitud $p-2$, etc. Implemente un algoritmo recursivo que imprima las marcas entre dos posiciones enteras, ambas incluidas. La longitud de las marcas para las posiciones enteras, p, es un parámetro. A la derecha se ve el resultado esperado para $p = 4$.
Note que la cantidad de marcas es $2 + (2^{p-1} - 1)$.

```
----
-
--
-
---
-
--
-
----
```