# Ciencias sociales computacionales

Práctica 4 – Funciones recursivas

**Aclaraciones**:

- Puede utilizar funciones auxiliares para resolver cualquiera de los ejercicios, siempre y cuando las mismas no tengan ciclos.
- Puede utilizar algún ítem anterior para resolver un ejercicio en caso de que corresponda.

## Ejercicio 1

Implemente las siguientes funciones sobre enteros en `R` utilizando recursión:

a) `fact(n)`: devuelve el valor del factorial de `n`.

b) `fib(n)`: devuelve el resultado de calcular el término `n`-ésimo de la sucesión de Fibonacci.

c) `sumatoriaPotenciasDeDos(n1, n2)`: devuelve el resultado de calcular $\sum_{i=\texttt{n1}}^{\texttt{n2}} 2^i$.

d) `sumaImpares(n)`: dado un `n` natural, devuelve la suma de todos los números naturales impares menores a él.

e) `sumaDigitos(n)`: devuelve la suma de todos los dígitos decimales de `n`. Por ejemplo, `sumaDigitos(56) == 11`

f) `sumaDivisores(n)`: devuelve la suma de todos los divisores positivos de `n`.

g) `divisiblePor3(n)`: devuelve `True` si `n` es divisible por 3. Para calcular esto, utilice el criterio de divisibilidad definido de la siguiente manera:

   - Un número de un dígito es divisible por 3 si es 3, 6 o 9.
   - Un número de más de un dígito es divisible por 3 si la suma de sus dígitos lo es.

h) `divisiblePor17(n)`: devuelve `True` si `n` es divisible por 17. El criterio es el siguiente:

   - Un número menor a 17 no es divisible por 17.
   - 17 es divisible por 17.
   - Si es mayor, se separa la última cifra de la derecha y se la multiplica por 5.
   - Se resta al resto de los dígitos el número obtenido.
   - Si el resultado es divisible por 17, entonces `n` es divisible por 17.

## Ejercicio 2

Implemente las siguientes funciones sobre listas en `R` utilizando recursión:

a) `suma(a)`: devuelve la suma de todos los elementos de la lista `a`.

b) `maximo(a)`: devuelve el máximo entre todos los elementos de la lista `a`.

c) `promedio(a)`: devuelve el promedio de todos los elementos de la lista `a`.

d) `listaDeAbs(a)`: devuelve una lista con los valores absolutos de cada elemento de la lista `a`.

e) `maximoAbsoluto(a)`: devuelve el máximo entre los valores absolutos de todos los elementos de la lista `a`.

f) `cambioDeBase(n, x)`: devuelve una lista donde cada elemento corresponde a la representación en base `x` del número indicado por `n`. Por ejemplo, `cambioDeBase(256, 2) == [1, 0, 0, 0, 0, 0, 0, 0, 0]`

g) `cantidadApariciones(a, x)`: devuelve la cantidad de veces que aparece el elemento `x` en la lista `a`.

h) `eliminar(a, i)`: devuelve una lista que tiene los mismos elementos que `a`, pero sin el elemento `i`-ésimo.

i) `buscarYEliminar(a, x)`: devuelve una lista que tiene los mismos elementos que `a`, pero donde se eliminaron todas las apariciones de `x`.

j) `todosPares(a)`: devuelve `True` si todos los elementos de la lista `a` son pares.

k) `ordenAscendente(a)`: devuelve `True` si todos los elementos de la lista `a` aparecen en orden ascendente. Por ejemplo, `ordenAscendente([1,2,4]) == True`

l) `reverso(a)`: devuelve una lista que cumple que sus elementos son los mismos que los de `a`, pero se encuentran en el orden inverso. Por ejemplo:

- `reverso(c('h','o','l','a')) == c('a','l','o','h')`
- `reverso(reverso(c('h','o','l','a'))) == c('h','o','l','a')`

m) `sumaPosImpares(a)`: devuelve el resultado de sumar todos los elementos de `a` que se encuentren en posiciones impares. Por ejemplo: `sumaPosImpares(c(3,5,4,3,2)) == 5 + 3`

n) `triangular(a)`: devuelve `True` si la lista `a` es creciente hasta cierto valor `m` y decreciente a partir de ese punto. Por ejemplo:

- `triangular(c(1,2,3,3,3)) == True`
- `triangular(c(1,2,1,2)) == False`
- `triangular(c(1)) == True`

**Ejercicio 3**

`R` permite que no solamente variables sino también funciones sean pasadas como parámetro a otras funciones. La manera de hacer esto es utilizando el nombre de la función sin los paréntesis. Esto permite abstraer patrones comunes en ciertos algoritmos y así evitar repetir código innecesariamente. Muchos de los algoritmos recursivos de esta práctica responden a *esquemas de recursión* comunes que ya forman parte de lenguaje. Implemente los siguientes esquemas de recursión en `R`:

a) `map(fn, a)`: toma una función unaria `fn` y una lista `a` y devuelve una nueva lista del mismo tamaño que la original, donde cada elemento resulta de aplicar la función `fn` a cada elemento de `a`. Por ejemplo, `map(fact, c(1,3,4)) == c(fact(1),fact(3),fact(4)) == c(1,6,24)`.

b) `filter(fn, a)`: toma una función unaria `fn` que devuelve un booleano y una lista `a` y devuelve una nueva lista que tiene únicamente los elementos `x` de `a` tales que `fn(x)` es verdadero. Por ejemplo, `filter(divisiblePor3, c(1,2,3,4,5,6)) == c(3,6)`.

c) (**Opcional**) `reduce(fn, a)`: toma una función binaria `fn` y una lista `a` y devuelve el resultado de `fn(a[1], reduce(fn, a[-1]))`. En caso de que `a` tenga un único elemento, debe devolver `a[1]`. Por ejemplo, `reduce(suma2, c(1,2,3,4)) == 10`, donde `suma2(x, y)` es una función que devuelve la suma entre `x` e `y`.

d) (**Opcional**) Piense cómo implementaría las funciones `suma`, `maximo`, `cantidadApariciones`, `todosPares` y `ordenAscendente` del ejercicio 2 utilizando alguna combinación de los esquemas de recursión presentados en los ítems anteriores.

**Ejercicio 4**

Implemente funciones en `R` utilizando recursión y que cumplan con las siguientes especificaciones.

a) problema $A(c(a : \mathbb{Z}), j : \mathbb{Z}) = res : \mathbb{Z}\{$
   requiere : $0 \leq j < |a|$;
   asegura : $res = \sum_{i=0}^{j} signo(a[i]) \cdot |a[i]^i|$;
   aux $signo(n : \mathbb{Z}) : \mathbb{Z} = (-1) \cdot \beta(n < 0) + \beta(n > 0)$;
   }

b) problema $B(n_1, n_2 : \mathbb{Z}) = res : \mathbb{Z}\{$
   requiere : $n_1 < n_2 \wedge (\exists k : \mathbb{Z})(n_1 < k \leq n_2 \wedge esPrimo(k))$;
   asegura : $n_1 < res \leq n_2 \wedge esPrimo(res) \wedge (\forall k : \mathbb{Z})(n_1 < k < res \rightarrow \neg esPrimo(k))$;
   }

c) problema $C(c(A, B : \mathbb{Z})) = c(sal : \mathbb{Z})\{$
   requiere : $|B| > 0$;
   asegura : $|sal| = |A| \wedge (\forall i : \mathbb{Z})\,(0 \leq i < |A| \rightarrow sal[i] = (A[i] + B[i \bmod |B|]))$;
   }

d) problema $D(c(X, V : \mathbb{Z})) = c(sal : \mathbb{Z})\{$
   asegura : $(\forall i : \mathbb{Z})(0 \leq i < |X| \rightarrow ((pert(X[i], sal) \wedge \neg pert(X[i], V)) \vee (\neg pert(X[i], sal) \wedge pert(X[i], V))) \wedge 0 \leq |sal| \leq |X|$;
   aux $pert(a : \mathbb{Z}, c(T : \mathbb{Z})) = (\exists j : \mathbb{Z})(0 \leq j < |T| \wedge T[j] = a)$;
   }

**Ejercicio 5**

(**Opcional**) El siguiente ejercicio corresponde a los temas de técnicas algorítmicas de la segunda parte de la materia. Dichos contenidos no forman parte de los temas del parcial.

Implemente las siguientes funciones sobre listas en `R` utilizando la técnica de Divide & Conquer:

a) `busquedaBinariaAcotada(a, i, j, x)`: devuelve `True` si el valor `x` se encuentra en la lista `a[seq(i,j)]`, sabiendo que esta última se encuentra ordenada. No utilice la notación de *slices* (`a[seq(i,j)]`) de `R`. El algoritmo implementado debe tener complejidad $O(\log n)$.

b) `mergeSort(a)`: devuelve una lista con los mismos elementos que `a`, ordenada de forma creciente por medio del algoritmo merge sort.

c) `quickSort(a)`: devuelve una lista con los mismos elementos que `a`, ordenada de forma creciente por medio del algoritmo quick sort.

d) `minimoRotado(a)`: devuelve el valor del mínimo elemento de la lista `a`, sabiendo que la lista se encuentra ordenada ascendentemente y rotada. Se dice que una lista $b$ se encuentra rotada con respecto a $a$ si se cumple que $|a| = |b| \wedge (\exists k : \mathbb{Z})(0 \leq k < |a| \wedge (\forall i : \mathbb{Z})(0 \leq i < |b| \rightarrow b[i] = a[(k + i) \bmod |a|]))$. Por ejemplo, `minimoRotado(c(8,10,1,4,4,5)) == 1`. El algoritmo implementado debe tener complejidad $O(\log n)$.

**Ayuda**: utilice una variante de la búsqueda binaria para resolver este problema.