Ejercicios TP3

Ejercicio 1: Sumar los elementos pares de un vector de números de dimensión N. No puede usar una función Par()

```
Funcion SumarParesIterativo(Vector, N)
    Suma ← 0
    Para i ← 1 Hasta N Hacer
        Si Vector[i] MOD 2 = 0 Entonces
            Suma ← Suma + Vector[i]
        Fin Si
    Fin Para
    Retornar Suma
Fin Funcion
```

```
Funcion SumarParesRecursivo(Vector, N)
    Si N = 0 Entonces
        Retornar 0
    Fin Si

    Si Vector[N] MOD 2 = 0 Entonces
        Retornar Vector[N] + SumarParesRecursivo(Vector, N - 1)
    Si No
        Retornar SumarParesRecursivo(Vector, N - 1)
    Fin Si
Fin Funcion
```

Ejercicio 2: Calcular la potencia *n* de un número *x*.

```
Funcion PotenciaIterativa(x, n)
    Si n = 0 Entonces
        Retornar 1
    Fin Si

    Resultado ← 1
    Para i ← 1 Hasta n Hacer
        Resultado ← Resultado * x
    Fin Para

    Retornar Resultado
Fin Funcion
```

Este algoritmo utiliza un bucle para multiplicar el número **x** consigo mismo **n** veces. En el caso especial en el que **n = 0**, se devuelve directamente **1** (por definición matemática).

```
Funcion PotenciaRecursiva(x, n)
    Si n = 0 Entonces
        Retornar 1
    Fin Si

    Retornar x * PotenciaRecursiva(x, n - 1)
Fin Funcion
```

Divide el problema en pasos más pequeños multiplicando el número **x** por el resultado de la potencia de **x** elevada a **n-1**, hasta alcanzar el caso base, cuando **n = 0**.

Ejercicio 3: Si un vector de dimensión N, formado por caracteres, indicar si el mismo es una palabra de tipo palíndromo.

```
FUNCION es_vector_palindromo_iterativo(vector):
    N <- LONGITUD(vector)
    PARA i = 1 HASTA N DIVIDIDO POR 2:
        SI vector[i] != vector[N - 1]:
            RETORNAR FALSO
    RETORNAR VERDADERO
```

```
FUNCION es_vector_palindromo_recursivo(vector):
    SI LONGITUD(vector) <= 1:
        RETORNAR VERDADERO
    SI vector[0] != vector[LONGITUD(vector) - 1]:
        RETORNAR FALSO
    RETORNAR es_vector_palindromo_recursivo(SUBVECTOR(vector, 1,
LONGITUD(vector) - 1))
```

Este algoritmo utiliza dos punteros, uno al inicio y otro al final del vector, comparando los caracteres hasta llegar al centro.

```
// Declaración de variables
CARACTER[] vector          // Vector de caracteres de dimensión N
ENTERO inicio              // Índice inicial (1)
ENTERO fin                 // Índice final (N)

FUNCION es_palindromo_iterativo(vector: CARACTER[]): BOOLEANO
    inicio <- 1
    fin <- LONGITUD(vector)   // Considerando que la última
posición es N
    MIENTRAS inicio < fin HACER
        SI vector[inicio] != vector[fin] ENTONCES
            RETORNAR FALSO
        FIN SI
        inicio <- inicio + 1
        fin <- fin - 1
    FIN MIENTRAS
    RETORNAR VERDADERO
FIN FUNCION
```

```
// Declaración de variables
CARACTER[] vector          // Vector de caracteres de dimensión N
ENTERO inicio              // Índice inicial (1)
ENTERO fin                 // Índice final (N)

FUNCION es_palindromo_recursivo(vector: CARACTER[], inicio:
ENTERO, fin: ENTERO): BOOLEANO
    SI inicio >= fin ENTONCES
        RETORNAR VERDADERO
    FIN SI
    SI vector[inicio] != vector[fin] ENTONCES
        RETORNAR FALSO
    FIN SI
    RETORNAR es_palindromo_recursivo(vector, inicio + 1, fin - 1)
FIN FUNCION
```

Explicación de los Algoritmos

1. **Iterativo:**
   - Compara los caracteres desde los extremos hacia el centro utilizando un bucle mientras.
   - Es eficiente en términos de memoria porque evita las llamadas recursivas.
   - Ideal para vectores grandes.

2. **Recursivo:**
   - Divide el problema en subproblemas más pequeños y compara los extremos en cada llamada.
   - Es más elegante y claro en su lógica, pero consume más memoria debido a la pila de llamadas recursivas.
   - Adecuado para vectores pequeños.

Prueba de Escritorio

Ejemplo: vector = ['r', 'a', 'd', 'a', 'r']

Algoritmo Iterativo

Paso	vector	inicio	fin	vector[inicio]	vector[fin]	Resultado parcial	Descripción
Inicial	['r', 'a', 'd', 'a', 'r']	1	5	'r'	'r'	VERDADERO	Comienza la comparación.
1	['r', 'a', 'd', 'a', 'r']	2	4	'a'	'a'	VERDADERO	Comparación en curso.
2	['r', 'a', 'd', 'a', 'r']	3	3	'd'	'd'	VERDADERO	Índices se encuentran.
Final	['r', 'a', 'd', 'a', 'r']	-	-	-	-	VERDADERO	El vector es palíndromo.

Algoritmo Recursivo

Paso	vector	inicio	fin	vector[inicio]	vector[fin]	Retorno parcial	Descripción
Llamada inicial	['r', 'a', 'd', 'a', 'r']	1	5	'r'	'r'	es_palindromo_recursivo(2, 4)	Llamada recursiva en curso.
Llamada recursiva	['r', 'a', 'd', 'a', 'r']	2	4	'a'	'a'	es_palindromo_recursivo(3, 3)	Llamada recursiva en curso.
Llamada recursiva	['r', 'a', 'd', 'a', 'r']	3	3	'd'	'd'	VERDADERO	Caso base alcanzado.
Resolviendo	['r', 'a', 'd', 'a', 'r']	-	-	-	-	VERDADERO	El vector es palíndromo.

Conclusión

Ambos algoritmos cumplen con la función de verificar si el vector es un palíndromo. Sin embargo:

1. **Iterativo**:
   - **Ventajas**:
     - Mayor eficiencia en memoria porque evita el uso de la pila de llamadas.
     - Más rápido en ejecución, especialmente para vectores grandes.
   - **Desventajas**:
     - Puede ser menos intuitivo para algunos casos.
2. **Recursivo**:
   - **Ventajas**:
     - Código más elegante y fácil de entender en términos de estructura.
   - **Desventajas**:
     - Consumo de memoria adicional debido a la pila de llamadas recursivas.
     - Puede ser menos eficiente para valores grandes de **N**.

Ejercicio 4: De un vector de dimensión N generar dos vectores, uno formado con elementos pares y otro con elementos impares.

Este algoritmo recorre el vector con un bucle y clasifica los elementos en pares e impares.

```
// Declaración de variables
ENTERO[] vector         // Vector de entrada con dimensión N
ENTERO[] pares          // Vector para almacenar elementos pares
ENTERO[] impares        // Vector para almacenar elementos impares
ENTERO i               // Índice del bucle
ENTERO elemento        // Elemento actual del vector

FUNCION clasificar_iterativo(vector: ENTERO[]): (ENTERO[],
ENTERO[])
    pares <- []
    impares <- []
    PARA i <- 1 HASTA LONGITUD(vector) HACER
        elemento <- vector[i]
        SI elemento MOD 2 == 0 ENTONCES
            AGREGAR elemento A pares
        SINO
            AGREGAR elemento A impares
        FIN SI
    FIN PARA
    RETORNAR (pares, impares)
FIN FUNCION
```

Este algoritmo divide el problema en subproblemas más pequeños. Clasifica el primer elemento y llama recursivamente con el resto del vector.

```
// Declaración de variables
ENTERO[] vector         // Vector de entrada con dimensión N
ENTERO elemento         // Elemento actual del vector

FUNCION clasificar_recursivo(vector: ENTERO[], i: ENTERO): (ENTERO[],
ENTERO[])
    SI i > LONGITUD(vector) ENTONCES
        RETORNAR ([], [])
    FIN SI
    elemento <- vector[i]
    (pares, impares) <- clasificar_recursivo(vector, i + 1)
    SI elemento MOD 2 == 0 ENTONCES
        AGREGAR elemento A pares
    SINO
        AGREGAR elemento A impares
    FIN SI
    RETORNAR (pares, impares)
FIN FUNCION
```

Prueba de Escritorio

```
Ejemplo: vector = [4, 7, 9, 2]
```

Algoritmo Iterativo

Paso	vector	i	elemento	pares	impares	Descripción
Inicial	[4, 7, 9, 2]	1	-	[]	[]	Vectores iniciales vacíos.
1	[4, 7, 9, 2]	1	4	[4]	[]	`4` es par, se agrega a `pares`.
2	[4, 7, 9, 2]	2	7	[4]	[7]	`7` es impar, se agrega a `impares`.
3	[4, 7, 9, 2]	3	9	[4]	[7, 9]	`9` es impar, se agrega a `impares`.
4	[4, 7, 9, 2]	4	2	[4, 2]	[7, 9]	`2` es par, se agrega a `pares`.
Final	[4, 7, 9, 2]	-	-	[4, 2]	[7, 9]	Vectores clasificados.

Algoritmo Recursivo

Paso	vector	i	elemento	pares	impares	Descripción
Llamada inicial	[4, 7, 9, 2]	1	4	[]	[]	Llama recursivamente con `i = 2`.
Llamada recursiva	[4, 7, 9, 2]	2	7	[4]	[]	Llama recursivamente con `i = 3`.
Llamada recursiva	[4, 7, 9, 2]	3	9	[4]	[7]	Llama recursivamente con `i = 4`.
Llamada recursiva	[4, 7, 9, 2]	4	2	[4]	[7, 9]	Llama recursivamente con `i = 5`.
Caso base	[]	5	-	[4, 2]	[7, 9]	Retorna vectores clasificados.

Explicación de cómo resuelve cada algoritmo el problema

1. **Iterativo**:
   - Utiliza un bucle para recorrer el vector desde la posición **1** hasta **N**.
   - Evalúa si cada elemento es par o impar y lo agrega al vector correspondiente.
   - Es eficiente en términos de memoria y ejecución, adecuado para manejar grandes volúmenes de datos.
2. **Recursivo**:
   - Divide el problema en partes más pequeñas, evaluando el primer elemento y luego llamándose a sí mismo con el resto del vector.
   - Es más intuitivo en términos de estructura de código, pero consume más memoria debido a las llamadas recursivas.
   - Puede ser menos eficiente en vectores grandes debido al límite de recursión en algunos lenguajes.

Ejercicio 5: Dado un vector de caracteres contar los elementos que sean letras del alfabeto.

Algoritmo Iterativo

El algoritmo recorre el vector con un bucle y verifica si cada carácter es una letra utilizando sus valores ASCII.

```
// Declaración de variables
CARACTER[] vector          // Vector de caracteres de dimensión N
ENTERO contador            // Contador de letras
CARACTER elemento          // Elemento actual del vector
ENTERO i                   // Índice del bucle

FUNCION contar_letras_iterativo(vector: CARACTER[]): ENTERO
    contador <- 0
    PARA i <- 1 HASTA LONGITUD(vector) HACER
        elemento <- vector[i]
        SI (ORDEN_ASCII(elemento) >= ORDEN_ASCII('A') Y
ORDEN_ASCII(elemento) <= ORDEN_ASCII('Z')) O
           (ORDEN_ASCII(elemento) >= ORDEN_ASCII('a') Y
ORDEN_ASCII(elemento) <= ORDEN_ASCII('z')) ENTONCES
            contador <- contador + 1
        FIN SI
    FIN PARA
    RETORNAR contador
FIN FUNCION
```

Algoritmo Recursivo

El algoritmo verifica si el primer elemento es una letra y llama recursivamente con el resto del vector.

```
// Declaración de variables
CARACTER[] vector          // Vector de caracteres de dimensión N
CARACTER elemento          // Elemento actual del vector

FUNCION contar_letras_recursivo(vector: CARACTER[], i: ENTERO): ENTERO
    SI i > LONGITUD(vector) ENTONCES
        RETORNAR 0
    FIN SI
    elemento <- vector[i]
    SI (ORDEN_ASCII(elemento) >= ORDEN_ASCII('A') Y ORDEN_ASCII(elemento)
<= ORDEN_ASCII('Z')) O
       (ORDEN_ASCII(elemento) >= ORDEN_ASCII('a') Y ORDEN_ASCII(elemento)
<= ORDEN_ASCII('z')) ENTONCES
        RETORNAR 1 + contar_letras_recursivo(vector, i + 1)
    SINO
        RETORNAR contar_letras_recursivo(vector, i + 1)
    FIN SI
FIN FUNCION
```

Prueba de Escritorio

```
Ejemplo: vector = ['a', '1', '#', 'B', 'z']
```

Algoritmo Iterativo

Paso	vector	i	elemento	ASCII	contador	Descripción
Inicial	['a', '1', '#', 'B', 'z']	1	-	-	0	Contador inicializado en 0.
1	['a', '1', '#', 'B', 'z']	1	'a'	97	1	'a' está en el rango ASCII 97-122.
2	['a', '1', '#', 'B', 'z']	2	'1'	49	1	'1' no está en ningún rango.
3	['a', '1', '#', 'B', 'z']	3	'#'	35	1	'#' no está en ningún rango.
4	['a', '1', '#', 'B', 'z']	4	'B'	66	2	'B' está en el rango ASCII 65-90.
5	['a', '1', '#', 'B', 'z']	5	'z'	122	3	'z' está en el rango ASCII 97-122.
Final	['a', '1', '#', 'B', 'z']	-	-	-	3	Fin del bucle, contador total = 3.

Algoritmo Recursivo

Paso	vector	i	elemento	ASCII	Retorno parcial	Descripción
Llamada inicial	['a', '1', '#', 'B', 'z']	1	'a'	97	1 + contar_letras_recursivo(vector, 2)	'a' está en el rango ASCII 97-122.
Llamada recursiva	['a', '1', '#', 'B', 'z']	2	'1'	49	contar_letras_recursivo(vector, 3)	'1' no está en ningún rango.
Llamada recursiva	['a', '1', '#', 'B', 'z']	3	'#'	35	contar_letras_recursivo(vector, 4)	'#' no está en ningún rango.
Llamada recursiva	['a', '1', '#', 'B', 'z']	4	'B'	66	1 + contar_letras_recursivo(vector, 5)	'B' está en el rango ASCII 65-90.
Llamada recursiva	['a', '1', '#', 'B', 'z']	5	'z'	122	1 + contar_letras_recursivo(vector, 6)	'z' está en el rango ASCII 97-122.
Caso base	[]	6	-	-	0	Fin de recursión, retorna 0.
Resolviendo	-	-	-	-	3	Suma total: `1 + 1 + 1 = 3`.

Explicación de cómo resuelve cada algoritmo el problema

1. **Iterativo**:
   - Utiliza un bucle para recorrer el vector desde la posición **1** hasta **N**.
   - Evalúa si cada elemento es una letra comparando su valor ASCII con los rangos válidos.
   - Es eficiente en términos de memoria y ejecución, adecuado para manejar grandes volúmenes de datos.

2. **Recursivo**:
   - Divide el problema en partes más pequeñas, verificando el primer elemento y luego llamándose a sí mismo con el resto del vector.
   - Es más intuitivo en términos de estructura de código, pero consume más memoria debido a las llamadas recursivas.
   - Puede ser menos eficiente en vectores grandes debido al límite de recursión en algunos lenguajes.

Ejercicio 6: Dado un Número entero Positivo, determinar la cantidad de Divisores del mismo.

```
// Declaración de variables
ENTERO n              // Número entero positivo
ENTERO i              // Iterador del bucle
ENTERO contador       // Contador de divisores

FUNCION contar_divisores_iterativo(n: ENTERO): ENTERO
    contador <- 0
    PARA i <- 1 HASTA n HACER
        SI n MOD i == 0 ENTONCES
            contador <- contador + 1
        FIN SI
    FIN PARA
    RETORNAR contador
FIN FUNCION

// Declaración de variables
ENTERO n              // Número entero positivo
ENTERO i              // Índice actual (inicia en 1)
```

```
FUNCION contar_divisores_recursivo(n: ENTERO, i: ENTERO): ENTERO
    SI i > n ENTONCES
        RETORNAR 0
    FIN SI
    SI n MOD i == 0 ENTONCES
        RETORNAR 1 + contar_divisores_recursivo(n, i + 1)
    SINO
        RETORNAR contar_divisores_recursivo(n, i + 1)
    FIN SI
FIN FUNCION
```

Prueba de Escritorio

```
Ejemplo: n = 6
```

Algoritmo Iterativo

Paso	n	i	n MOD i	contador	Descripción
Inicial	6	-	-	0	Contador inicializado en 0.
1	6	1	0	1	6 MOD 1 == 0, se incrementa.
2	6	2	0	2	6 MOD 2 == 0, se incrementa.
3	6	3	0	3	6 MOD 3 == 0, se incrementa.
4	6	4	2	3	6 MOD 4 != 0, no se incrementa.
5	6	5	1	3	6 MOD 5 != 0, no se incrementa.
6	6	6	0	4	6 MOD 6 == 0, se incrementa.
Final	6	-	-	4	Fin del bucle, contador = 4.

Algoritmo Recursivo

Paso	n	i	n MOD i	Retorno parcial	Descripción
Llamada inicial	6	1	0	1 + contar_divisores_recursivo(6, 2)	6 MOD 1 == 0, suma 1.
Llamada recursiva	6	2	0	1 + contar_divisores_recursivo(6, 3)	6 MOD 2 == 0, suma 1.
Llamada recursiva	6	3	0	1 + contar_divisores_recursivo(6, 4)	6 MOD 3 == 0, suma 1.
Llamada recursiva	6	4	2	contar_divisores_recursivo(6, 5)	6 MOD 4 != 0, no suma.
Llamada recursiva	6	5	1	contar_divisores_recursivo(6, 6)	6 MOD 5 != 0, no suma.
Llamada recursiva	6	6	0	1 + contar_divisores_recursivo(6, 7)	6 MOD 6 == 0, suma 1.
Caso base	6	7	-	0	Fin de recursión, retorna 0.
Resolviendo	6	-	-	4	Suma total: 1 + 1 + 1 + 1 = 4.

Explicación de cómo resuelve cada algoritmo el problema

1. **Iterativo**:
   - Utiliza un bucle para recorrer el vector desde la posición **1** hasta **N**.
   - Evalúa si cada elemento es una letra comparando su valor ASCII con los rangos válidos.
   - Es eficiente en términos de memoria y ejecución, adecuado para manejar grandes volúmenes de datos.
2. **Recursivo**:
   - Divide el problema en partes más pequeñas, verificando el primer elemento y luego llamándose a sí mismo con el resto del vector.
   - Es más intuitivo en términos de estructura de código, pero consume más memoria debido a las llamadas recursivas.
   - Puede ser menos eficiente en vectores grandes debido al límite de recursión en algunos lenguajes.

OTROS EJERCICIOS

Determinar si un número es primo

Un número N es primo si sólo es divisible por 1 y por sí mismo

1. Si el número es menor o igual a 1, no es primo.
2. Si el número es 2, es primo.
3. Si el número es divisible por 2 (es par), no es primo.
4. Para todos los números impares desde 3 hasta la raíz cuadrada del número:
    - Si el número es divisible por alguno de estos números, no es primo.
5. Si no se encontró ningún divisor, el número es primo.

```
// Declaración de variables
ENTERO n              // Número entero positivo
ENTERO i              // Iterador para el bucle

FUNCION es_primo_iterativo(n: ENTERO): BOOLEANO
    SI n <= 1 ENTONCES
        RETORNAR FALSO
    FIN SI
    PARA i <- 2 HASTA n - 1 HACER
        SI n MOD i == 0 ENTONCES
            RETORNAR FALSO
        FIN SI
    FIN PARA
    RETORNAR VERDADERO
FIN FUNCION
```

```
// Declaración de variables
ENTERO n              // Número entero positivo
ENTERO i              // Iterador para las llamadas recursivas
(inicia en 2)

FUNCION es_primo_recursivo(n: ENTERO, i: ENTERO): BOOLEANO
    SI n <= 1 ENTONCES
        RETORNAR FALSO
    FIN SI
    SI i >= n ENTONCES
        RETORNAR VERDADERO
    FIN SI
    SI n MOD i == 0 ENTONCES
        RETORNAR FALSO
    FIN SI
    RETORNAR es_primo_recursivo(n, i + 1)
FIN FUNCION
```

Prueba de Escritorio

```
Ejemplo: n = 7
```

Algoritmo Iterativo

Paso	n	i	n MOD i	Resultado parcial	Descripción
Inicial	7	-	-	-	Comienza la evaluación.
1	7	2	1	VERDADERO	7 MOD 2 != 0, sigue el bucle.
2	7	3	1	VERDADERO	7 MOD 3 != 0, sigue el bucle.
3	7	4	3	VERDADERO	7 MOD 4 != 0, sigue el bucle.
4	7	5	7	VERDADERO	7 MOD 5 != 0, sigue el bucle.
Final	7	-	-	VERDADERO	Retorna VERDADERO, es primo.

Algoritmo Recursivo

Paso	n	i	n MOD i	Resultado parcial	Descripción
Llamada inicial	7	2	1	es_primo_recursivo(7, 3)	7 MOD 2 != 0, llamada recursiva.
Llamada recursiva	7	3	1	es_primo_recursivo(7, 4)	7 MOD 3 != 0, llamada recursiva.
Llamada recursiva	7	4	3	es_primo_recursivo(7, 5)	7 MOD 4 != 0, llamada recursiva.
Llamada recursiva	7	5	7	es_primo_recursivo(7, 6)	7 MOD 5 != 0, llamada recursiva.
Llamada recursiva	7	6	-	VERDADERO	i >= n, retorna VERDADERO.

Determinar el factorial de un número n

```
// Declaración de variables
ENTERO n         // Número del que se calculará el factorial
ENTERO resultado // Almacena el resultado del cálculo
ENTERO i         // Iterador para el bucle

FUNCION factorial_iterativo(n: ENTERO): ENTERO
    resultado <- 1
    PARA i <- 1 HASTA n HACER
        resultado <- resultado * i
    FIN PARA
    RETORNAR resultado
FIN FUNCION

// Ejemplo de uso
n <- 5
IMPRIMIR "Factorial: ", factorial_iterativo(n)  // Salida: 120
```

```
// Declaración de variables
ENTERO n         // Número del que se calculará el factorial

FUNCION factorial_recursivo(n: ENTERO): ENTERO
    SI n == 0 ENTONCES
        RETORNAR 1
    FIN SI
    RETORNAR n * factorial_recursivo(n - 1)
FIN FUNCION

// Ejemplo de uso
n <- 5
IMPRIMIR "Factorial: ", factorial_recursivo(n)  // Salida: 120
```

Prueba de escritorio: Algoritmo Iterativo

Ejemplo: Factorial de n = 4

Paso	n	i	resultado	Descripción
Inicial	4	-	1	Se inicializa resultado = 1.
1	4	1	1	resultado = 1 * 1.
2	4	2	2	resultado = 1 * 2.
3	4	3	6	resultado = 2 * 3.
4	4	4	24	resultado = 6 * 4.
Final	4	-	24	Termina el bucle y retorna resultado = 24.

Prueba de escritorio: Algoritmo Recursivo

Paso	n	Retorno	Descripción
Llamada inicial	4	4 * f(3)	Calcula 4 * factorial_recursivo(3).
Llamada recursiva	3	3 * f(2)	Calcula 3 * factorial_recursivo(2).
Llamada recursiva	2	2 * f(1)	Calcula 2 * factorial_recursivo(1).
Llamada recursiva	1	1 * f(0)	Calcula 1 * factorial_recursivo(0).
Caso base	0	1	Retorna 1 como caso base.
Resolviendo	-	1 * 1 = 1	Retroceso: resultado parcial.
Resolviendo	-	2 * 1 = 2	Retroceso: resultado parcial.
Resolviendo	-	3 * 2 = 6	Retroceso: resultado parcial.
Resolviendo	-	4 * 6 = 24	Retroceso final: retorna 24.