**EJERCICIOS DE VECTORES**

1. **Dado un vector con diferentes valores imprimir dichos valores del vector leyéndolos uno por uno de manera recursiva.**

#lang racket

(define (imprimir-vector-recursivo vec)

; Definimos una función recursiva para imprimir el vector

(define (imprimir-recursivo idx)

; Comprobamos si el índice es menor que la longitud del vector

(if (< idx (vector-length vec))

(begin

(display (vector-ref vec idx)) ; Imprime el valor en la posición idx

(newline) ; Imprime un salto de línea

(imprimir-recursivo (+ idx 1)) ; Llamada recursiva al siguiente índice

) ; Cierra el 'begin'

(void) ; Si el índice es igual o mayor, termina la recursión

) ; Cierra el 'if'

) ; Cierra la definición de 'imprimir-recursivo'

; Llamamos a la función recursiva empezando desde el índice 0

(imprimir-recursivo 0)

)

; Función para leer n valores del usuario y almacenarlos en un vector

(define (leer-valores n)

(define valores '()) ; Lista vacía para almacenar los valores

(define (leer-valores-recursivo idx)

(if (< idx n)

(begin

(display "Ingrese un valor: ")

(let ([val (read)]) ; Usamos let para definir val dentro de la función

(set! valores (append valores (list val)))) ; Agregamos el valor a la lista

(leer-valores-recursivo (+ idx 1)) ; Llamamos recursivamente

)

(vector (apply vector valores)) ; Convertimos la lista a un vector

)

)

(leer-valores-recursivo 0) ; Llamamos a la función recursiva comenzando desde el índice 0

)

; Pedimos al usuario el tamaño del vector

(display "Ingrese el tamaño del vector: ")

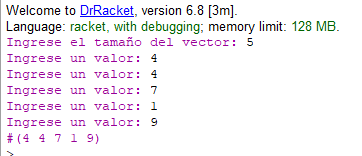
(define n (read))

; Leemos los valores

(define mi-vector (leer-valores n))

; Imprimimos el vector ingresado

(imprimir-vector-recursivo mi-vector)



1. **Dado un número n, crear un vector de tamaño n y luego ingresar en el vector los números del 1 hasta n, e imprimir el vector. Ej: Dado el número 4 ingresar en el vector (vector 1 2 3 4).**

#lang racket

; Función para crear el vector y llenarlo con los números del 1 hasta n

(define (crear-vector n)

(define (llenar-vector idx vec)

(if (< idx n) ; Aseguramos que el índice sea menor que n

(begin

(vector-set! vec idx (+ idx 1)) ; Establecemos el valor como idx + 1 (porque el índice comienza en 0)

(llenar-vector (+ idx 1) vec) ; Llamada recursiva para llenar el siguiente índice

)

vec)) ; Devolvemos el vector cuando hemos llegado a n

(let ([vec (make-vector n)]) ; Creamos un vector vacío de tamaño n

(llenar-vector 0 vec) ; Llenamos el vector comenzando desde el índice 0

vec)) ; Devolvemos el vector lleno

; Función para imprimir el vector

(define (imprimir-vector vec)

(define (imprimir-recursivo idx)

(if (< idx (vector-length vec))

(begin

(display (vector-ref vec idx)) ; Imprime el valor en la posición idx

(display " ") ; Agrega un espacio para separar los números

(imprimir-recursivo (+ idx 1)) ; Llamada recursiva al siguiente índice

)

(newline))) ; Salto de línea al finalizar la impresión

(imprimir-recursivo 0)) ; Comenzamos desde el índice 0

; Solicitar al usuario el tamaño del vector

(display "Ingrese el tamaño del vector: ")

(define n (read))

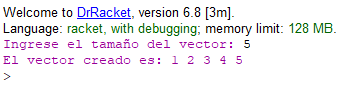
; Crear el vector con números del 1 hasta n

(define mi-vector (crear-vector n))

; Imprimir el vector

(display "El vector creado es: ")

(imprimir-vector mi-vector)



1. **Llenar un vector V de 10 elementos con cuadrados de los 10 primeros elementos. Ejemplo: (vector 1 4 9 1 6 25 36 49 64 81 100).**

#lang racket

; Función para crear un vector de tamaño n y llenarlo con los cuadrados de los números del 1 hasta n

(define (crear-vector-cuadrados n)

(define (llenar-vector idx vec)

(if (< idx n) ; Aseguramos que el índice esté dentro del rango

(begin

(vector-set! vec idx (\* (+ idx 1) (+ idx 1))) ; Asignamos el cuadrado de (idx + 1)

(llenar-vector (+ idx 1) vec)) ; Llamada recursiva al siguiente índice

vec)) ; Devolvemos el vector cuando hemos llenado todos los elementos

(let ([vec (make-vector n)]) ; Creamos un vector vacío de tamaño n

(llenar-vector 0 vec) ; Llenamos el vector comenzando desde el índice 0

vec)) ; Devolvemos el vector lleno

; Función para imprimir el vector

(define (imprimir-vector vec)

(define (imprimir-recursivo idx)

(if (< idx (vector-length vec))

(begin

(display (vector-ref vec idx)) ; Imprime el valor en la posición idx

(display " ") ; Espacio entre los números

(imprimir-recursivo (+ idx 1)) ; Llamada recursiva al siguiente índice

)

(newline))) ; Salto de línea al finalizar la impresión

(imprimir-recursivo 0)) ; Comenzamos desde el índice 0

; Solicitar al usuario el tamaño del vector

(display "Ingrese el tamaño del vector (debe ser 10 o más): ")

(define n (read))

; Verificar que el tamaño sea al menos 10

(if (< n 10)

(display "El tamaño debe ser al menos 10. Se establecerá un tamaño de 10.")

(set! n 10))

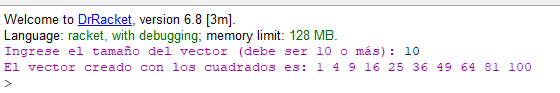
; Crear el vector con los cuadrados de los números del 1 hasta n

(define mi-vector (crear-vector-cuadrados n))

; Imprimir el vector

(display "El vector creado con los cuadrados es: ")

(imprimir-vector mi-vector)



1. **Dado un vector V de enteros y un número X, devolver el valor que corresponde al número de veces que está el valor X en el vector.**

#lang racket

; Función para contar cuántas veces aparece el número X en el vector

(define (contar-valor-en-vector vec x)

; Función recursiva para recorrer el vector y contar las veces que aparece X

(define (contar-recursivo idx count)

(if (< idx (vector-length vec)) ; Si el índice es menor que la longitud del vector

(if (= (vector-ref vec idx) x) ; Si el valor en la posición idx es igual a X

(contar-recursivo (+ idx 1) (+ count 1)) ; Aumentamos el contador

(contar-recursivo (+ idx 1) count)) ; No aumentamos el contador

count)) ; Devolvemos el contador cuando llegamos al final del vector

(contar-recursivo 0 0)) ; Comenzamos desde el índice 0 y el contador en 0

; Función para leer un vector de tamaño n ingresado por el usuario

(define (leer-vector n)

(define vec (make-vector n)) ; Creamos el vector vacío de tamaño n

; Función recursiva para llenar el vector con los valores ingresados

(define (leer-recursivo idx)

(if (< idx n) ; Mientras el índice sea menor que el tamaño del vector

(begin

(display "Ingrese el valor para el índice ")

(display idx)

(display ": ")

(let ((valor (read))) ; Leemos el valor usando let para evitar 'define' en expresión

(vector-set! vec idx valor)) ; Asignamos el valor al vector

(leer-recursivo (+ idx 1)) ; Llamamos a la función recursiva para el siguiente índice

)

vec)) ; Devolvemos el vector cuando se haya llenado

(leer-recursivo 0)) ; Comenzamos desde el índice 0

; Ejemplo de uso

; Solicitar al usuario el tamaño del vector

(display "Ingrese el tamaño del vector: ")

(define n (read))

; Crear un vector de tamaño n y llenarlo con los valores ingresados por el usuario

(define mi-vector (leer-vector n))

; Mostrar el vector ingresado

(display "El vector ingresado es: ")

(display mi-vector)

(newline)

; Solicitar al usuario el valor X para contar cuántas veces aparece en el vector

(display "Ingrese el valor de X: ")

(define x (read))

; Contar cuántas veces aparece X en el vector

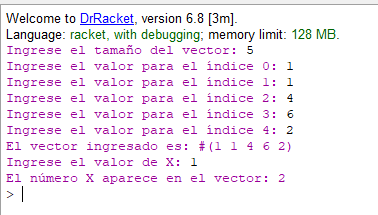
(define resultado (contar-valor-en-vector mi-vector x))

; Mostrar el resultado

(display "El número X aparece en el vector: ")

(display resultado)

(newline)



**EJERCICIOS DE LABORATORIO**

1. **Construir una función que reciba un parámetro. Si el parámetro es un carácter devolver el número que corresponda en la tabla del código ASCII y si es un número devolver el carácter que corresponda en la tabla. Tomar en cuenta que la función (number? n), retorna verdadero si n es un número y falso de lo contrario.**

#lang racket

(define (ascii-converter input)

(cond

[(number? input) (integer->char input)] ; Si es número, convertir a carácter.

[(char? input) (char->integer input)] ; Si es carácter, convertir a código ASCII.

[else "Entrada inválida"])) ; Para cualquier otro caso.

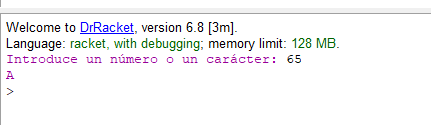
; Solicitar datos al usuario:

(display "Introduce un número o un carácter: ")

(define user-input (read)) ; Leer entrada del usuario.

; Mostrar el resultado:#\A 65 HOLA SALE INVALIDO

(displayln (ascii-converter user-input))



1. **Construir un programa que permita recibir una cadena y contar cuantos espacios en blanco tiene (emplear recursividad).**

#lang racket

(define (contar-espacios cadena)

(define (recursivo pos contador)

(if (>= pos (string-length cadena)) ; Caso base: si hemos llegado al final de la cadena.

contador

(recursivo (+ pos 1) ; Avanzar al siguiente carácter.

(if (char=? (string-ref cadena pos) #\space) ; Si es un espacio, aumentar contador.

(+ contador 1)

contador))))

(recursivo 0 0)) ; Iniciar la recursión desde la posición 0 con contador en 0.

; Solicitar cadena al usuario:

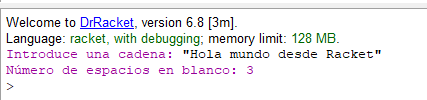
(display "Introduce una cadena: ")

(define user-input (read-line)) ; Leer cadena del usuario.

; Contar espacios y mostrar resultado:

(define resultado (contar-espacios user-input))

(displayln (string-append "Número de espacios en blanco: " (number->string resultado)))



1. **Construir un programa que reciba un vector de enteros y un número X. Se requiere que el programa busque el número x en el vector y devuelva la posición donde se encuentra por primera vez en el vector. En caso de no estar debe devolver #f**

#lang racket

; Función para buscar el número en el vector

(define (buscar-en-vector vector x)

(define len (vector-length vector)) ; Calcula la longitud del vector

(define (buscar-posicion i)

(cond

[(= i len) #f] ; Si llega al final del vector sin encontrar x, devuelve #f

[(= (vector-ref vector i) x) i] ; Si encuentra x, devuelve la posición i

[else (buscar-posicion (+ i 1))])) ; Llama recursivamente para la siguiente posición

(buscar-posicion 0)) ; Inicia la búsqueda desde la posición 0

; Solicitar al usuario los valores del vector

(display "Ingrese el número de elementos del vector: ")

(define n (read)) ; Leer la cantidad de elementos

; Crear un vector vacío y llenarlo con valores proporcionados por el usuario

(define mi-vector (make-vector n))

(for ([i n])

(display (string-append "Ingrese el elemento " (number->string (+ i 1)) ": "))

(vector-set! mi-vector i (read)))

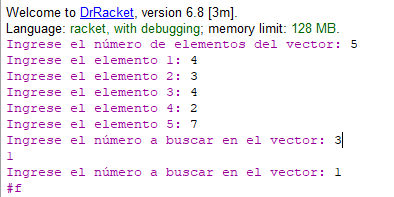
; Realizar dos búsquedas consecutivas

(for ([j 2])

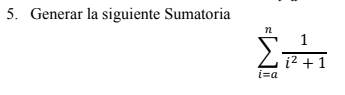
(display "Ingrese el número a buscar en el vector: ")

(define x (read)) ; Leer el número a buscar

(displayln (buscar-en-vector mi-vector x))) ;



**4. Generar la siguiente Sumatoria**



#lang racket

(define (sumatoria a n)

(define (suma acumulador i)

(if (> i n)

acumulador

(suma (+ acumulador (/ 1 (+ (\* i i) 1))) (+ i 1))))

(suma 0 a))

; Solicitar el valor inicial a

(display "Ingrese el valor inicial (a): ")

(define a (read))

; Solicitar el valor final n

(display "Ingrese el valor final (n): ")

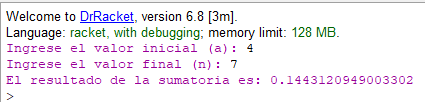
(define n (read))

; Calcular la sumatoria y mostrar el resultado en formato decimal

(define resultado (sumatoria a n))

(display "El resultado de la sumatoria es: ")

(displayln (exact->inexact resultado)) ; Convierte



**EJERCICIOS DE LABORATORIO DEL OTRO LAB**

1. **Construya una función que reciba una cadena y la devuelva invertida**

#lang racket

(define (cadenaInvertida)

(display "Introduzca una cadena: ") ; Solicita la entrada al usuario.

(define cadena (read-line)) ; Lee toda la línea como una cadena.

(define lon (string-length cadena)) ; Calcula la longitud de la cadena.

(display "Cadena invertida: ") ; Imprime el encabezado de la salida.

(displayln (revertir cadena (- lon 1) "")) ; Imprime la cadena invertida.

; Devuelve la cadena invertida como resultado de la función (opcional).

(revertir cadena (- lon 1) ""))

(define (revertir cadena pos nuevo)

(if (< pos 0) ; Si se alcanza la posición inicial (< 0),

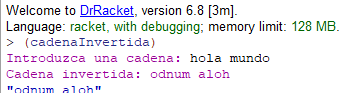
nuevo ; devolver la cadena invertida.

(revertir cadena ; De lo contrario, continuar recursión:

(- pos 1) ; disminuir posición.

(string-append nuevo ; añadir carácter actual al resultado.

(string (string-ref cadena pos))))))



1. **Construir un programa que reciba un vector de enteros y obtenga el promedio de los números del vector. Necesariamente utilizar recursividad**

#lang racket

; Función principal

(define (promedio-vector)

(display "Introduce los números del vector separados por espacios: ")

(define entrada (read-line)) ; Leer una línea de entrada como cadena.

(define numeros ; Convertir la cadena en una lista de números.

(map string->number (string-split entrada)))

(define suma (sumar-lista numeros)) ; Llama a la función recursiva para sumar.

(define cantidad (length numeros)) ; Obtiene la cantidad de números.

(if (= cantidad 0) ; Verifica si el vector está vacío.

(displayln "El vector está vacío. No se puede calcular el promedio.")

(displayln (string-append

"El promedio es: "

(number->string (/ suma cantidad))))))

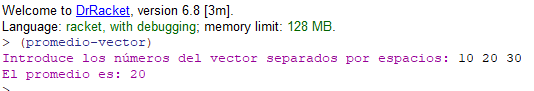
; Función recursiva para sumar los elementos de la lista.

(define (sumar-lista lista)

(if (null? lista) ; Caso base: si la lista está vacía.

0 ; La suma es 0.

(+ (first lista) (sumar-lista (rest lista))))) ; Sumar primer elemento con el resto.



1. **Genera la sumatoria Σ^n \_(i=a)(1/i^3+1)**

#lang racket

; Función principal para solicitar los límites de la sumatoria.

(define (calcular-sumatoria)

(display "Introduce el valor de a (límite inferior): ")

(define a (read)) ; Leer el límite inferior.

(display "Introduce el valor de n (límite superior): ")

(define n (read)) ; Leer el límite superior.

(if (> a n)

(displayln "Error: el límite inferior (a) no puede ser mayor que el límite superior (n).")

(let ((resultado (sumatoria a n))) ; Usamos 'let' para definir la variable de resultado.

(displayln (string-append

"La sumatoria es: "

(number->string resultado)))))) ; Mostrar el resultado.

; Función recursiva para calcular la sumatoria.

(define (sumatoria i n)

(if (> i n) ; Caso base: si el índice supera el límite superior, la suma termina.

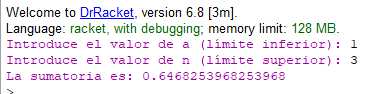
0.0 ; Retornar 0.0 como número decimal.

(+ (/ 1.0 (+ (expt i 3) 1)) ; Sumar el término actual: 1 / (i^3 + 1) en decimal.

(sumatoria (+ i 1) n)))) ; Llamada recursiva para el siguiente índice.

; Ejecutar la función para probar.

(calcular-sumatoria)



1. **Escribe un programa que cuente el número de vocales y consonantes que contiene necesariamente usar recursividad**

#lang racket

; Función principal para solicitar los límites de la sumatoria.

(define (calcular-sumatoria)

(display "Introduce el valor de a (límite inferior): ")

(define a (read)) ; Leer el límite inferior.

(display "Introduce el valor de n (límite superior): ")

(define n (read)) ; Leer el límite superior.

(if (> a n)

(displayln "Error: el límite inferior (a) no puede ser mayor que el límite superior (n).")

(let ((resultado (sumatoria a n))) ; Usamos 'let' para definir la variable de resultado.

(displayln (string-append

"La sumatoria es: "

(number->string resultado)))))) ; Mostrar el resultado.

; Función recursiva para calcular la sumatoria.

(define (sumatoria i n)

(if (> i n) ; Caso base: si el índice supera el límite superior, la suma termina.

0.0 ; Retornar 0.0 como número decimal.

(+ (/ 1.0 (+ (expt i 3) 1)) ; Sumar el término actual: 1 / (i^3 + 1) en decimal.

(sumatoria (+ i 1) n)))) ; Llamada recursiva para el siguiente índice.

; Ejecutar la función para probar.

(calcular-sumatoria)

