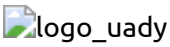
TLP



Teoría de Lenguajes de Programación LIS/LCC

TLP - Proyecto

Profesor: M.C. Luis R. Basto

- [16003312] Llanes Montero, Roberto Carlos (100%)
- [10003294] Bustamante Lara, Rigel David (0%)

Fecha de entrega: 09/02/2022

Instalación

Para empezar, instala el jupyter notebook siguiendo el tutorial que se presenta en su página Jupyter notebook

Además, agrega el iRacket, el kernel necesario para que puedas usar racket con los jupyter notebooks: iRacket.

Ahora, usando python crea un entorno virtual para instalar las dependencias.

```
python -m venv env
```

Una vez creado el entorno virtual, activalo usando el siguiente comando:

```
source ./env/bin/active
```

Por último, instala las dependencias necesarias con ayuda de pip y el archivo de requerimients.txt.

```
pip install -r requirements.txt
```

Si quieres salir del entorno virtual, puedes usar el comando `deactivate`:

Usando el notebook

Para iniciar el notebook

```
jupyter notebook
```

El notebook esta dividido en secciones para cada ejercicio, cada sección tiene:

1. La descriptión del problema.
2. El código de la solución en racket.
3. Los casos de prueba para cada problema, cada uno con una pequeña descripción en markdown y la ejecución de la función.

Al final de todas las secciones, hay un pequeño espacio donde puedes probar todas las funciones ya cargadas.

Ejecución de los problemas

Ejemplo 01

1. Defina la función B que calcule los coeficientes binomiales B(n,k) = n!/((n-k)!*k! para n>=0, y 0<=k<=n

In [5]:
```
#|
B -> number
parametros ->
    n: número de elementos de un conjunto.
    k: número de elementos a escoger del conjunto n.
Devuelve el número de formas en que se puede extraer subconjuntos a partir de un conjunto dado.
|#
(define (factorial n)
  (cond
    [(= n 0) 1]
    [(= n 1) 1]
    [else
     (* n (factorial (- n 1)))]
))
(define (division a b)(/ a b))
(define (multiplicacion a b)(* a b))
(define (resta a b)(- a b))
(define (B n k)(division(factorial n)
                        (multiplicacion (factorial (resta n k)) (factorial k))))
```

Casos de prueba 1.

n -> 5

k -> 3

Resultado esperado: 10

In [6]:
```
(B 5 3)
```

Out[6]:
```
10
```

Ejemplo 02

2. Defina una función para la evaluación del número combinatorio C(n,k), que utiliza la definición recursiva.

In [9]:
```
#|
C -> number
parametros ->
    n: número de elementos de un conjunto.
    k: número de elementos a escoger del conjunto n.
Devuelve el número de combinaciones.
|#

(define (factorial n)
  (cond
    [(= n 0) 1]
    [(= n 1) 1]
    [else
     (* n (factorial (- n 1)))]
))

(define (combinacion a b)
        (cond
    [(= b 0) 1]
    [(= a b) 1]
    [else
     (division(factorial a)
                       (multiplicacion (factorial (resta a b)) (factorial b)))]))

(define (division a b)(/ a b))
(define (multiplicacion a b)(* a b))
(define (suma a b)(+ a b))
(define (resta a b)(- a b))
(define (C n k)(suma(combinacion (- n 1)(- k 1))
                       (combinacion (- n 1)(+ k 0) )))
```

Casos de prueba 1.

n -> 5

k -> 3

Resultado esperado: 10

In [10]:
```
(C 5 3)
```

Out[10]:
```
10
```

Ejemplo 03

3. Defina una función recursiva para calcular el Máximo Común Divisor de dos enteros negativos a y b con a < b usando el hecho de que MCD(a, b) = MCD(a, b-a).

In [121]:
```
#|
MDC -> number
parametros ->
    a: primer numero
    b: Segundo número.
Devuelve el máximo común.
|#
(define (mcd a b)
  (if (> b a)
      (mcd b a)
      (if (= (modulo a b) 0) b (mcd b (modulo a b)))))
```

Casos de prueba 1.

a -> 100

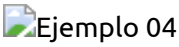
b -> 10

Resultado esperado: 10

In [122]:
```
(mcd 100 10)
```

Out[122]:
```
10
```

Ejemplo 04


Ejemplo 04

Ejemplo 05

5. Realizar una función para hallar el valor de e definiéndola como sigue:

In [62]:
```
(define (factorial n)
  (cond
    [(= n 0) 1]
    [(= n 1) 1]
    [else
     (* n (factorial (- n 1)))]
))
(define (division a b)(/ a b))
(define (suma a b)(+ a b))
#|
euler -> number
parametros ->
    a: valor a evaluar en la función.
Devuelve el resultado de evaluar la función de euler para una valor a.
|#
(define (euler a)
  (if (= a 0)
      1
      (suma(division 1 (factorial a))(euler (- a 1)))
  ))
```

Casos de prueba 1.

a -> 1

Resultado esperado: 2

In [63]:
```
(euler 6)
```

Out[63]: 1957/720

Ejemplo 06

6. Realiza una función que indique la longitud de una lista.

In [69]:
```
#|
len -> number
parametros ->
    lista: lista de los elementos a contar.
Devuelve el número de elementos de una lista.
|#
(define (len lista)
  (if (null? lista)
      0
      (+ 1 (len (cdr lista)))))
```

Casos de prueba 1.

lista -> '(1 2 3 4)

Resultado esperado: 4

In [70]:
```
(len '(1 2 3 4))
```

Out[70]: 4

Ejemplo 07

7. Realiza una función para buscar un elemento en una lista, regresar #t si lo encontró y #f si no lo encontró

In [93]:
```
#|
busqueda -> boolean
parametros ->
    elemento: elemento a buscar en la lista.
    lista: lista de elementos.
Devuelve #t en caso de que encuentre el valor en la lista, #f en caso contrario.
|#
(define (busqueda elemento lista) (if (null? lista) #f (if (= elemento (car lista)) #t (busqueda elemento (cdr lista)
```

Casos de prueba 1.

lista -> '()

elemento -> 100

Resultado esperado: #f

In [98]:
```
(busqueda 100 '(1 2))
```

Out[98]: #f

Ejemplo 08

8. Realizar una funcion que invierta una lista

In [101]:
```
#|
invertir -> list
parametros ->
    elemento: elemento a buscar en la lista.
    lista: lista de elementos.
Dada una lista, invierte sus valores.
|#
(define (invertir lista)
  (if (null? lista)
      '()
      (append (invertir (cdr lista)) (list (car lista)))))
```

Casos de prueba 1.

lista -> '()

Resultado esperado: '()

In [102]:
```
(invertir '())
```

Out[102]: '()

Casos de prueba 2.

lista -> '(1 2 2 1)

Resultado esperado: '(1 2 2 1)

In [103]:
```
(invertir '(1 2 2 1))
```

Out[103]: '(1 2 2 1)

Ejemplo 09

9. Realizar una función que elimine un elemento de una listas

In [105]:
```
#|
eliminar -> lista
parametros ->
    lista: lista de elementos.
    elemento: elemento a eliminar en la lista.
Eliminar todos los elementos de la lista que coincidan con el elemento a buscar.
|#
(define (eliminar lista n)
  (filter (lambda (elemento) (if (= elemento n) #f #t)) lista))
```

Casos de prueba 1.

lista -> '(1 2 3 4 4)

elemento -> 3

Resultado esperado: '(1 2 4 4)

In [106]:
```
(eliminar '(1 2 3 4 4) 3)
```

Out[106]:
```
'(1 2 4 4)
```

Casos de prueba 2.

lista -> '(1 2 3 4 4)

elemento -> 5

Resultado esperado: '(1 2 3 4 4)

In [107]:
```
(eliminar '(1 2 3 4 4) 5)
```

Out[107]:
```
'(1 2 3 4 4)
```

Ejemplo 10

10. Defina una función para calcular la desviación estándar de un conjunto de datos (lista) La fórmula es la siguiente:
Imagen de la formula de Desviacion estandar

In [125]:
```
;; Funciones auxiliares
(define (len lista) (if (null? lista) 0 (+ 1 (len (cdr lista)))))

(define (sumatoria elementos) (if (null? elementos) 0 (+ (car elementos) (sumatoria (cdr elementos)))))

(define (elevarAlCuadrado valor) (* valor valor))

(define (media datos)(/ (sumatoria datos) (len datos)))

(define (diferenciaConMedia valor media) (- valor media))

(define (aplicarDiferenciaAlCuadrado datos)
  (map (lambda (dato)
         (elevarAlCuadrado
          (diferenciaConMedia dato (media datos))))
       datos))

#|
desviacionEstandar -> numero
parametros ->
    datos: lista de datos que se usaran para calcular su desviacion estandar.
Dada una lista de datos, devuelve su desviacion estandar.
|#
(define (desviacionEstandar datos)
  (/
   (sumatoria
    (aplicarDiferenciaAlCuadrado datos))
   (len datos)))
```

Casos de prueba 1.

lista -> '(1 1 1 1)

Resultado esperado: 0

In [126]:
```
(desviacionEstandar '(1 1 1 1))
```

Out[126]:
```
0
```