Memoria P1
Inteligencia Artificial

Martín De Las Heras Moreno 05445941R
martin.delasheras@estudiante.uam.es
Santiago Valderrábano Zamorano 51500439N
santiago.valderrabano@estudiante.uam.es

CONTENIDO

Ejercicio 1:

Apartado 1:

- *PRODUCTO-ESCALAR-REC*

PSEUDOCODIGO:

Entrada: x, y dos vectores
Salida: producto escalar de los dos vectores

Procesamiento:
Si alguno de los dos vectores (o los dos) son null devuelve 0, en caso contrario devuelve su producto escalar.

CÓDIGO:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; producto-escalar-rec (x y)
;;; Calcula el producto escalar de dos vectores
;;; Se asume que los dos vectores de entrada tienen la misma longitud.
;;;
;;; INPUT: x: vector, representado como una lista
;;;        y: vector, representado como una lista
;;; OUTPUT: producto escalar de x e y
;;;

(defun producto-escalar-rec (x y)
  (if (or (null x) (null y))
    0
    (+ (* (first x) (first y)) (producto-escalar-rec (rest x) (rest y))))
 )
```

COMENTARIOS:

Esta función es una auxiliar para calcular la distancia coseno entre dos vectores de manera recursiva.

- *COSINE-DISTANCE*

PSEUDOCODIGO:

Entrada: x, y dos vectores
Salida: distancia coseno entre esos dos vectores.

Procesamiento:
Aplica la fórmula de la distancia coseno.

CÓDIGO:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; cosine-distance (x y)
;;; Aplica la formula de distancia coseno dados numerador y denominador
;;; de la fórmula
;;;
;;; INPUT:  x: numerador
;;;         y: denominador
;;; OUTPUT: distancia coseno entre x e y
;;;

(defun cosine-distance (x y)
   (- 1 (/ x y))
 )
```

COMENTARIOS:

Esta función es una auxiliar para calcular la distancia coseno entre dos vectores de manera recursiva.

- *COSINE-DISTANCE-REC*

PSEUDOCODIGO:

Entrada: x, y dos vectores
Salida: distancia coseno entre esos dos vectores.

Procesamiento:
Si alguno de los dos productos escalares de los vectores da 0 devuelve nil, si no devuelve la distancia coseno entre los dos vectores.

CÓDIGO:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; cosine-distance-rec (x y)
;;; Calcula la distancia coseno de un vector de forma recursiva
;;; Se asume que los dos vectores de entrada tienen la misma longitud.
;;;
;;; INPUT: x: vector, representado como una lista
;;;             y: vector, representado como una lista
;;; OUTPUT: distancia coseno entre x e y
;;;

(defun cosine-distance-rec (x y)
  (if (= 0 (* (producto-escalar-rec x x) (producto-escalar-rec y y)))
   nil
   (cosine-distance (producto-escalar-rec x y) (* (sqrt (producto-escalar-rec x x)) (sqrt (producto-
escalar-rec y y)))))
 )
```

COMENTARIOS:

Esta función es una auxiliar para calcular la distancia coseno entre dos vectores de manera recursiva.

- *COSINE-DISTANCE-MAPCAR*

PSEUDOCODIGO:

Entrada: x, y dos vectores
Salida: distancia coseno entre esos dos vectores.

Procesamiento:
Si alguno de los dos productos escalares de los vectores da 0 devuelve nil, si no devuelve la distancia coseno entre los dos vectores.

CÓDIGO:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; cosine-distance-mapcar (x y)
;;; Calcula la distancia coseno de un vector usando mapcar
;;; Se asume que los dos vectores de entrada tienen la misma longitud.
;;;
;;; INPUT: x: vector, representado como una lista
;;;            y: vector, representado como una lista
;;; OUTPUT: distancia coseno entre x e y
;;;

(defun cosine-distance-mapcar (x y)
  (if (= 0 (* (apply #'+ (mapcar #'* x x)) (apply #'+ (mapcar #'* y y))))
    0
    (cosine-distance (apply #'+ (mapcar #'* x y)) (* (sqrt (apply #'+ (mapcar #'* x x))) (sqrt (apply
#'+ (mapcar #'* y y))))))
  )
```

COMENTARIOS:

Esta función es una auxiliar para calcular la distancia coseno entre dos vectores usando mapcar.

Los resultados a las pruebas pedidas en el ejercicio 1.1 son:

1. (cosine-distance '(1 2) '(1 2 3)) → 0.40238577
2. (cosine-distance nil '(1 2 3)) → 0
3. (cosine-distance '() '()) → 0
4. (cosine-distance '(0 0) '(0 0)) → 0

Apartado 2:

En este apartado se pide crear una función que ordene los vectores de la lista según el valor de confianza con el vector de referencia, siempre y cuando superen el valor de confianza dado como argumento de entrada:

- *INSERT-ORDERED-COSINE-DISTANCE*

PSEUDOCODIGO:

Entrada: reference, vector, lst-of-vectors siendo estos el vector de referencia, el vector a insertar y la lista donde se inserta.
Salida: lista de vectores actualizada con el vector insertado.

Procesamiento:
Esta función recibe una referencia (el vector respecto del cual se ordena la lista) un vector y la lista para insertarlo, si el vector es nulo devuelve la lista como tal, si lo nulo es la lista devuelve el vector como único elemento de la lista, en caso de que el vector sea más cercano que la lista se inserta el primero y en caso contrario se pasa al siguiente por recursión.

CÓDIGO:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;  insert-ordered-cosine-distance
;;; Inserta en orden segun la semejanza un vector en una lista
;;; INPUT:  reference: vector de referencia para la semejanza
;;;         vector: vector a insertar
;;;         lst-of-vectors: lista de vectores donde se va a insertar el nuevo
;;; OUTPUT: Lista de vectores con el nuevo dentro
;;;

(defun insert-ordered-cosine-distance (reference vector lst-of-vectors)
  (cond ((null vector)
         lst-of-vectors)
        ((null lst-of-vectors)
         (cons vector lst-of-vectors))
        ((< (cosine-distance-mapcar reference vector) (cosine-distance-mapcar reference (first lst-of-vectors)))
         (cons vector lst-of-vectors))
        (t
         (cons (first lst-of-vectors) (insert-ordered-cosine-distance reference vector (rest lst-of-vectors)))))
  )
```

COMENTARIOS:

Inserta en orden de distancia un vector nuevo en la lista ya existente.

- *ORDER-VECTORS-COSINE-DISTANCE-REC*

PSEUDOCODIGO:

Entrada: vector, lst-of-vectors, confidence-level, siendo estos el vector de categoria, un vector de vectores y un nivel de confianza.
Salida: lista de vectores ordenada por distancia coseno.

Procesamiento:
Esta función envía a la anterior de manera recursiva todos los elementos de la lista para que los vaya ordenando.

CÓDIGO:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; order-vectors-cosine-distance-rec
;;; Parte recursiva de order-vectors-cosine-distance-rec
;;; INPUT:  vector: vector que representa a una categoria, representado como una lista
;;;              lst-of-vectors vector de vectores
;;;               confidence-level: Nivel de confianza (parametro opcional)
;;; OUTPUT: Vectores cuya semejanza con respecto a la
;;;         categoria es superior al nivel de confianza ,
;;;         ordenados
;;;

(defun order-vectors-cosine-distance-rec (vector lst-of-vectors &optional (confidence-level 0))
  (cond ((null lst-of-vectors)
         nil)
        (t
         (insert-ordered-cosine-distance vector (first lst-of-vectors) (order-vectors-cosine-distance-rec
vector (rest lst-of-vectors) confidence-level))))
  )
```

COMENTARIOS:
Parte recursiva de la función principal.

- *ORDER-VECTORS-COSINE-DISTANCE*

PSEUDOCODIGO:
Entrada: vector, lst-of-vectors, confidence-level, siendo estos el vector de categoria, un vector de vectores y un nivel de confianza.
Salida: lista de vectores ordenada por distancia coseno con los inferiores al nivel de confianza eliminados.

Procesamiento:
Esta función envía la lista de vectores a la recursiva y cuando le llega de vuelta elimina aquellos con una distancia inferior al nivel de confianza.

CÓDIGO:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; order-vectors-cosine-distance
;;; Devuelve aquellos vectores similares a una categoria
;;; INPUT:  vector: vector que representa a una categoria,
;;;             representado como una lista
;;;         lst-of-vectors vector de vectores
;;;         confidence-level: Nivel de confianza (parametro opcional)
;;; OUTPUT: Vectores cuya semejanza con respecto a la
;;;         categoria es superior al nivel de confianza ,
;;;         ordenados
;;;

(defun order-vectors-cosine-distance (vector lst-of-vectors &optional (confidence-level 0))
  (order-vectors-cosine-distance-rec vector
                              (remove-if (lambda (v) (< (- 1 confidence-level) (cosine-distance-mapcar
vector v))) lst-of-vectors)
                              confidence-level)
  )
```

COMENTARIOS:

Función principal y única llamada desde la línea de comandos.

La salida a los ejercicios puestos es:

(order-vectors-cosine-distance '(1 2 3) '()) → NIL
(order-vectors-cosine-distance '() '((4 3 2) (1 2 3))) → ((1 2 3) (4 3 2))

Apartado 3:

En este ejercicio se pide que dada una lista de textos y de categorías se le asigne a cada texto la categoría más cercana según la función de distancia asignada.
Para ello hemos creado dos funciones, la primera que dado un texto y una lista de categorías saque la que le corresponde al texto y la otra que dada una lista de textos y otra de categorías llame repetidamente a la primera para que saque las categorías.

- ***GET-CATEGORY***

PSEUDOCODIGO:

Entrada: categories, texts, distance-measure siendo estos la lista de categorías, el texto que hay que ubicar y la función usada para hallar la distancia.
Salida: par formado por el identificador de la categoría y el valor de la distancia entre esta y el texto.

Procesamiento:
Esta función recorre la lista de categorías y va actualizando el mínimo cuando encuentra uno más pequeño, cuando la lista de categorías es null se entiende que se han acabado y por tanto envía de vuelta el mínimo en ese momento.

CÓDIGO:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; get-category (categories text distance-measure)
;;; Clasifica al texto en su categoria
;;;
;;; INPUT : categories: vector de vectores, representado como
;;;                 una lista de listas
;;;         texts:     vector
;;;         distance-measure: funcion de distancia
;;; OUTPUT: Par formado por el vector que identifica la categoria
;;;         de menor distancia , junto con el valor de dicha distanciaNUMBER
;;;

(defun get-category (categories text distance-measure minimum)
  (cond ((null categories)
        (list (first (first minimum)) (second minimum)))
       (t
        (if (< (funcall distance-measure text (first categories)) (second minimum))
          (get-category (rest categories) text distance-measure (list (first categories) (funcall distance-
measure text (first categories))))
         (get-category (rest categories) text distance-measure minimum))))
  )
```

COMENTARIOS:

Función auxiliar de la siguiente.

- *GET-VECTORS-CATEGORY*

PSEUDOCODIGO:

Entrada: categories, texts, distance-measure siendo estos la lista de categorías, lista de textos que hay que ubicar y la función usada para hallar la distancia.
Salida: lista de pares formados por el identificador de la categoría y el valor de la distancia entre esta y cada texto.

Procesamiento:
Esta función recorre la lista de textos y va enviando cada uno a la función get-category y con los resultados va formando una lista de pares.

CÓDIGO:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; get-vectors-category (categories vectors distance-measure)
;;; Clasifica a los textos en categorias .
;;;
;;; INPUT : categories: vector de vectores, representado como
;;;                 una lista de listas
;;;         texts:     vector de vectores, representado como
;;;                 una lista de listas
;;;         distance-measure: funcion de distancia
;;; OUTPUT: Pares formados por el vector que identifica la categoria
;;;         de menor distancia , junto con el valor de dicha distancia
;;;
```

```
(defun get-vectors-category (categories texts distance-measure)
  (cond ((null texts)
         nil)
        (t
         (cons (get-category (rest categories) (first texts) distance-measure (list (first categories)
(funcall distance-measure (first texts) (first categories)))) (get-vectors-category categories (rest
texts) distance-measure))))
  )
```

COMENTARIOS:

Función principal que llama recursivamente tanto a sí misma como a get-category para cada texto.

Apartado 4:

(get-vectors-category '(()) '(()) #'cosine-distance-mapcar) → **(NIL 0)**
(get-vectors-category '((1 4 2) (2 1 2)) '((1 1 2 3)) #'cosine-distance-mapcar) → **(2 0.39753592)**
(get-vectors-category '(()) '((1 1 2 3) (2 4 5 6)) #'cosine-distance-mapcar) → **((NIL 0) (NIL 0))**

Ejercicio 2:

Apartado 1:

- *NEWTON*

PSEUDOCODIGO:

Entrada: f, df, max-iter, x0, tol siendo estos la función, su derivada, el número máximo de iteraciones, el punto inicial y la tolerancia.
Salida: estimación del 0 de f o nil si no converge.

Procesamiento:
Aplica el método de Newton-Raphson para encontrar un 0 en una función.

CÓDIGO:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; newton
;;; Estima el cero de una funcion mediante Newton-Raphson
;;;
;;; INPUT : f: funcion cuyo cero se desea encontrar
;;;         df: derivada de f
;;;         max-iter: maximo numero de iteraciones
;;;         x0: estimacion inicial del cero (semilla)
;;;         tol: tolerancia para convergencia (parametro opcional)
;;; OUTPUT: estimacion del cero de f o NIL si no converge
;;;

(defun newton (f df max-iter x0 &optional (tol 0.001))
  (cond ((= max-iter 0)
        nil)
       ((< (abs (funcall f x0)) tol)
        x0)
       (t
        (newton f df (- max-iter 1) (- x0 (/ (funcall f x0) (funcall df x0))) tol)))
  )
```

COMENTARIOS:

Calcula si converge o no y el punto donde lo hace.

Apartado 2:

- *ONE-ROOT-NEWTON*

PSEUDOCODIGO:

Entrada: f, df, max-iter, semillas, tol siendo estos la función, su derivada, el número máximo de iteraciones, los puntos iniciales y la tolerancia.
Salida: primera semilla con la cual converge f.

Procesamiento:
Aplica Newton con cada una de las semillas hasta que una converja.

CÓDIGO:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; one-root-newton
;;; Prueba con distintas semillas iniciales hasta que Newton
;;; converge
;;;
;;; INPUT: f : funcion de la que se desea encontrar un cero
;;;        df : derivada de f
;;;        max-iter : maximo numero de iteraciones
;;;        semillas : semillas con las que invocar a Newton
;;;        tol : tolerancia para convergencia ( parametro opcional )
;;;
;;; OUTPUT: el primer cero de f que se encuentre , o NIL si se diverge
;;;          para todas las semillas
;;;

(defun one-root-newton (f df max-iter semillas &optional (tol 0.001))
  (cond ((null semillas)
        nil)
       ((null (newton f df max-iter (first semillas) tol))
        (one-root-newton f df max-iter (rest semillas) tol))
       (t
        (newton f df max-iter (first semillas) tol)))
  )
```

COMENTARIOS:

Dada una lista de semillas prueba hasta que una converge.

Apartado 3:

- ***ALL-ROOT-NEWTON***

PSEUDOCODIGO:

Entrada: f, df, max-iter, x0, tol siendo estos la función, su derivada, el número máximo de iteraciones, el punto inicial y la tolerancia.
Salida: raices de las semillas que converjan o nil si no lo hace.

Procesamiento:
Aplica Newton con cada una de las semillas y devuelve una lista con las raíces de las semillas que convergen y nil en las que no lo hacen.

CÓDIGO:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; all-roots-newton
;;; Prueba con distintas semillas iniciales y devuelve las raices
;;; encontradas por Newton para dichas semillas
;;;
;;; INPUT: f: funcion de la que se desea encontrar un cero
;;;        df: derivada de f
;;;        max-iter: maximo numero de iteraciones
;;;        semillas: semillas con las que invocar a Newton
;;;        tol : tolerancia para convergencia ( parametro opcional )
;;;
;;; OUTPUT: las raices que se encuentren para cada semilla o nil
;;;          si para esa semilla el metodo no converge
;;;

(defun all-roots-newton (f df max-iter semillas &optional ( tol 0.001))
  (cond ((null semillas)
         nil)
        (t
         (cons (newton f df max-iter (first semillas) tol) (all-roots-newton f df max-iter (rest semillas)
tol))))
  )
```

COMENTARIOS:

Devuelve lista con todas las raíces que ha encontrado con Newton o nil.

- *LIST-NOT-NIL-ROOTS-NEWTON*

PSEUDOCODIGO:

Entrada: f, df, max-iter, x0, tol siendo estos la función, su derivada, el número máximo de iteraciones, el punto inicial y la tolerancia.
Salida: raices de las semillas que converjan.

Procesamiento:
Llama a all-roots-newton y elimina los nil de la lista que devuelve.

CÓDIGO:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; list-not-nil-roots-newton
;;; Elimina los nil de all-roots-newton
;;;
;;; INPUT: f: funcion de la que se desea encontrar un cero
;;;        df: derivada de f
;;;        max-iter: maximo numero de iteraciones
;;;        semillas: semillas con las que invocar a Newton
;;;        tol : tolerancia para convergencia ( parametro opcional )
;;;
;;; OUTPUT: las raices que se encuentren para cada semilla
;;;
```

```
(defun list-not-nil-roots-newton (f df max-iter semillas &optional ( tol 0.001))
  (mapcan #'(lambda (x) (if (null x) nil (list x))) (all-roots-newton f df max-iter semillas tol))
 )
```

COMENTARIOS:

Igual que all-roots-newton pero sin devolver los nil.

Ejercicio 3:

Apartado 1:

- *COMBINE-ELT-LST*

PSEUDOCODIGO:

Entrada: elt y lst siendo estos el elemento y la lista a combinar.
Salida: lista con las combinaciones del elemento con los de la lista.

Procesamiento:
Usando mapcar creamos un par con elt y cada elemento de lst.

CÓDIGO:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; combine-elt-lst
;;; Combina un elemento dado con todos los elementos de una lista
;;;
;;; INPUT: elem: elemento a combinar
;;;        lst: lista con la que se quiere combinar el elemento
;;;
;;; OUTPUT: lista con las combinacion del elemento con cada uno de
;;;         los de la lista

(defun combine-elt-lst (elt lst)
  (mapcar #'(lambda (x) (list elt x)) lst)
  )
```

COMENTARIOS:

Combina una lista con un elemento.

La salida para los casos propuestos es:

1. (combine-elt-lst 'a nil) → **NIL**
2. (combine-elt-lst nil nil) → **NIL**
3. (combine-elt-lst nil '(a b)) → **NIL**

Apartado 2:

- *COMBINE-LST-LST*

PSEUDOCODIGO:

Entrada: lst1 y lst2 siendo estos las dos listas a combinar.
Salida: lista con las combinaciones de las dos listas.

Procesamiento:
Llama por cada elemento de lst1 a combine-elt-lst.

CÓDIGO:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; combine-lst-lst
;;; Calcula el producto cartesiano de dos listas
;;;
;;; INPUT: lst1: primera lista
;;;        lst2: segunda lista
;;;
;;; OUTPUT: producto cartesiano de las dos listas

(defun combine-lst-lst (lst1 lst2)
  (if (null lst1)
    nil
      (append (combine-elt-lst (first lst1) lst2) (combine-lst-lst (rest lst1) lst2)))
  )
```

COMENTARIOS:

Combina dos listas entre sí.

La salida para los casos propuestos es:

1. (combine-lst-lst nil nil) → **NIL**
2. (combine-lst-lst '(a b c) nil) → **NIL**
3. (combine-lst-lst nil '(a b c)) → **NIL**

Apartado 3:

- *COMBINE-ELT-LST-CONS*

PSEUDOCODIGO:

Entrada: elt y lst siendo estos el elemento y la lista a combinar.
Salida: lista con las combinaciones del elemento con los de la lista.

Procesamiento:
Usando mapcar creamos un par con elt y cada elemento de lst usando cons.

CÓDIGO:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; combine-elt-lst-cons
;;; Combina un elemento dado con todos los elementos de una lista usando cons
;;;
;;; INPUT: elem: elemento a combinar
;;;        lst: lista con la que se quiere combinar el elemento
;;;
;;; OUTPUT: lista con las combinacion del elemento con cada uno de los
;;;         de la lista
```

```
(defun combine-elt-lst-cons (elt lst)
  (mapcar #'(lambda (x) (cons elt x)) lst)
 )
```

COMENTARIOS:
Combina una lista con un elemento.

- *COMBINE-LST-LST*

PSEUDOCODIGO:

Entrada: lst1 y lst2 siendo estos las dos listas a combinar.
Salida: lista con las combinaciones de las dos listas.

Procesamiento:
Llama por cada elemento de lst1 a combine-elt-lst-cons.

CÓDIGO:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; combine-lst-lst-cons
;;; Calcula el producto cartesiano de dos listas usando cons
;;;
;;; INPUT: lst1: primera lista
;;;        lst2: segunda lista
;;;
;;; OUTPUT: producto cartesiano de las dos listas

(defun combine-lst-lst-cons (lst1 lst2)
(if (null lst1)
  nil
   (append (combine-elt-lst-cons (first lst1) lst2) (combine-lst-lst-cons (rest lst1) lst2)))
 )
```

COMENTARIOS:
Combina dos listas entre sí.

- *COMBINE-LIST-OF-LSTS*

PSEUDOCODIGO:

Entrada: lstolsts lista de listas.
Salida: lista con todas las combinaciones posibles.

Procesamiento:
Llama de manera recursiva combine-lst-lst-cons y como argumento pone en lst1 a la primera lista de la lista de listas y como segundo a una llamada recursiva de sí misma con el resto de la lista de listas.

CÓDIGO:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; combine-list-of-lsts
;;; Calcula todas las posibles disposiciones de elementos
;;; pertenecientes a N listas de forma que en cada disposicion
;;; aparezca unicamente un elemento de cada lista
;;;
;;; INPUT: lstolsts: lista de listas
;;;
;;; OUTPUT: lista con todas las posibles combinaciones de elementos

(defun combine-list-of-lsts (lstolsts)
  (if (null lstolsts)
    (list nil)
    (combine-lst-lst-cons (first lstolsts) (combine-list-of-lsts (rest lstolsts))))
  )
```

COMENTARIOS:

Combina todas las listas de la lista entre sí.

La salida para los casos propuestos es:

1. (combine-list-of-lsts '(() (+ -) (1 2 3 4))) → **NIL**
2. (combine-list-of-lsts '((a b c) () (1 2 3 4))) →**NIL**
3. (combine-list-of-lsts '((a b c) (1 2 3 4) ())) → **NIL**
4. (combine-list-of-lsts '((1 2 3 4))) → **((1) (2) (3) (4))**
5. (combine-list-of-lsts '(nil)) → **NIL**
6. (combine-list-of-lsts nil) → (**NIL)**

Ejercicio 4:

Apartado 1:

Para este apartado hemos implementado los siguientes grupos de funciones:

1. **Grupo 1:** Funciones para evaluar expresiones que comienzan con una negación.

- *EVALUAR-NEG-AND*

PSEUDOCODIGO:

Entrada: fbf (Fórmula bien formada)
Salida: fbf (Fórmula bien formada negada)

Procesamiento:
Si el resto es null
Devuelve lista que contiene el primer elemento con un not (!) delante
En otro caso crea una lista con un not y el primer elemento y se llama a si misma con el resto de la lista.

CÓDIGO:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; evaluar-neg-and
;;; Recibe una expresion y niega sus terminos
;;;
;;; INPUT  : fbf - Formula bien formada (FBF) a analizar cuyo primer
;;;              conector es ! ^
;;; OUTPUT : fbf  - Lista con los argumentos atomicos negados
;;;

(defun evaluar-neg-and (fbf)
  (if (null (rest fbf))
    (list (list +not+ (first fbf)))
    (append (list (list +not+ (first fbf))) (evaluar-neg-and (rest fbf)))))
```

COMENTARIOS:
Niega cada uno de los términos de la fbf que se le pasa como argumento.

- *EVALUAR-NEG-OR*

PSEUDOCODIGO:

Entrada: fbf (Fórmula bien formada)
Salida: fbf (Fórmula bien formada negada)
Procesamiento:
Si el resto es null
Devuelve lista que contiene el primer elemento con un not (!) delante
En otro caso crea una lista con un not y el primer elemento y se llama a si misma con el resto de la lista.

CÓDIGO:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; evaluar-neg-or
;;; Recibe una expresion y niega sus términos
;;;
;;; INPUT  : fbf - Formula bien formada (FBF) a analizar cuyo primer
;;;              conector es ! v
;;; OUTPUT : fbf - Lista con los argumentos atomicos negados
;;;

(defun evaluar-neg-or (fbf)
  (if (null (rest fbf))
    (list (list +not+ (first fbf)))
    (append (list (list +not+ (first fbf))) (evaluar-neg-or (rest fbf)))))
```

COMENTARIOS:
Niega cada uno de los términos de la fbf que se le pasa como argumento.

- *EVALUAR-N-ARY-NEG*

PSEUDOCODIGO:

Entrada: fbf (Fórmula bien formada)
Salida: fbf (Fórmula bien formada negada)

Procesamiento:
Si el primer elemento de fbf es V (or)
Devuelve lista que contiene un ^ (and) al principio y lo que devuelve negar-neg-or del resto
Si el primer elemento de fbf es ^ (and)
Devuelve lista que contiene un V (or) al principio y lo que devuelve negar-neg-and del resto

CÓDIGO:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; evaluar-n-ary-neg
;;; Recibe una expresion y evalua a que función
;;; mandar el resto de la fbf dependiendo del
;;; conector que preceda
;;;
;;; INPUT  : fbf - Formula bien formada (FBF) a analizar
;;; OUTPUT : fbf - Lista con los argumentos atomicos
;;;
(defun evaluar-n-ary-neg (fbf)
  (cond ((eql (first fbf) +or+)
    (append (list +and+) (evaluar-neg-or (rest fbf))))
   ((eql (first fbf) +and+)
    (append (list +or+) (evaluar-neg-and (rest fbf))))
    nil)
```

COMENTARIOS:
Dependiendo el conector que tenga manda el resto de fbf a negar-or o negar-and y lo que devuelven estas funciones lo añaden a una lista con el conector opuesto al que tenía la fbf original.

- *EVALUAR-NEG-COND*

PSEUDOCODIGO:

Entrada: fbf (Fórmula bien formada)
Salida: fbf (Fórmula bien formada negada)

Procesamiento:
Si es null
Devuelve nil.
En otro caso aplica la fórmula para negar una condicional.

CÓDIGO:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; evaluar-neg-cond
;;; Recibe una expresion y la niega sabiendo que el conector que la
;;  precedia era ! =>
;;;
;;; INPUT  : fbf - Formula bien formada (FBF) a analizar cuyo primer
;;;              conector es ! =>
;;; OUTPUT : fbf - Lista con la fbf negada
;;;
(defun evaluar-neg-cond (fbf)
(if (null fbf)
  nil
  (list +and+ (list +not+ (second fbf)) (first fbf))))
```

COMENTARIOS:
Aplica la fórmula para negar una expresión condicional.

- *EVALUAR-NEG-BICOND*

PSEUDOCODIGO:

Entrada: fbf (Fórmula bien formada)
Salida: fbf (Fórmula bien formada negada)

Procesamiento:
Aplica la fórmula para negar una bicondicional.

CÓDIGO:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; evaluar-neg-bicond
;;; Recibe una expresion y la niega sabiendo que el conector que la
;;  precedia era ! <=>
;;;
;;; INPUT  : fbf - Formula bien formada (FBF) a analizar cuyo primer
;;;              conector es ! <=>
;;; OUTPUT : fbf - Lista con la fbf negada
;;;

(defun evaluar-neg-bicond (fbf)
  (list +or+ (evaluar-neg-cond fbf) (evaluar-neg-cond (list (second fbf) (first fbf)))))
```

COMENTARIOS:
Aplica la fórmula para negar una expresión bicondicional.

- *EVALUAR-NOT*

PSEUDOCODIGO:

Entrada: fbf (Fórmula bien formada)
Salida: fbf (Fórmula bien formada negada)

Procesamiento:
Si el primer elemento es not (!)
Devuelve el resto de la fbf (doble negación)
Si el primer elemento es un conector n-ario
Manda toda la fbf a la función evaluar-n-ary-neg
Si el primer elemento es <=>
Manda el resto de la fbf a evaluar-neg-bicond
Si el primer elemento es =>
Manda el resto de la fbf a evaluar-neg-cond
En otro caso devuelve la lista con un not (!) delante

CÓDIGO:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; evaluar-not
;;; Recibe una expresion y la evalua
;;;
;;; INPUT  : fbf - Formula bien formada (FBF) a analizar cuyo primer
;;;              conector es ! y evalua a que funcion la tiene que
;;;              mandar dependiendo del siguiente conector.
;;; OUTPUT : list - Lista con la fbf negada
;;;

(defun evaluar-not (fbf)
  (cond ((unary-connector-p (first fbf))
        (rest fbf))
    ((n-ary-connector-p (first fbf))
        (evaluar-n-ary-neg fbf))
    ((bicond-connector-p (first fbf))
        (evaluar-neg-bicond (rest fbf)))
    ((cond-connector-p (first fbf))
        (evaluar-neg-cond (rest fbf)))
    (t
     (list +not+ (first fbf)))))
```

COMENTARIOS:
Recibe una fbf la cual va a mandar a una función de negación dependiendo del primer elemento que tenga.

2. **Grupo 2:** Funciones que convierten las expresiones que comienzan con un conector binario en una expression con conectores n-arios.

- *EVALUAR-COND*

PSEUDOCODIGO:

Entrada: fbf (Fórmula bien formada)
Salida: fbf (Fórmula bien formada con conectores n-arios)

Procesamiento:
Si es null lo que recibe
Devuelve nil
En otro caso devuelve lo que resulta de aplicar la fórmula de conversión de una expresión condicional a una expresión con conectores n-arios

CÓDIGO:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; evaluar-cond
;;; Recibe una expresion y la evalua
;;;
;;; INPUT  : fbf - Formula bien formada (FBF) a analizar cuyo primer
;;;              conector es =>
;;; OUTPUT : fbf - Lista con la expresion con solo conectores n-arios
;;;          NIL  - En caso de que los elementos sean vacios o NIL
;;;

(defun evaluar-cond (fbf)
  (if (null fbf)
    nil)
  (list +or+ (list +not+ (first fbf)) (second fbf)))
```

COMENTARIOS:
Aplica la fórmula para convertir una expresión condicional en una que solo contenga ands, ors y negaciones.

- *EVALUAR-BICOND*

PSEUDOCODIGO:

Entrada: fbf (Fórmula bien formada)
Salida: fbf (Fórmula bien formada con conectores n-arios)

Procesamiento:
Devuelve lo que resulta de aplicar la fórmula de conversión de una expresión condicional a una expresión con conectores n-arios

CÓDIGO:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; evaluar-cond
;;; Recibe una expresion y la evalua
;;;
;;; INPUT  : fbf - Formula bien formada (FBF) a analizar cuyo primer
;;;              conector es <=>
;;; OUTPUT : fbf - Lista con la expresion con solo conectores n-arios
;;;

(defun evaluar-bicond (fbf)
  (list +and+ (evaluar-cond fbf) (evaluar-cond (list (second fbf) (first fbf)))))
```

COMENTARIOS:
Aplica la fórmula para convertir una expresión bicondicional en una que solo contenga ands, ors y negaciones.

- *EVALUAR*

PSEUDOCODIGO:

Entrada: fbf (Fórmula bien formada)
Salida: fbf (Fórmula bien formada con conectores n-arios)

Procesamiento:
Si la fbf es un literal
Devuelve la fbf tal cual le llega
Si el primer elemento es not (!)
Manda a evaluar lo que le devuelva la función evaluar-not de lo que esta negado
Si el primer elemento es <=>
Manda a evaluar lo que devuelva el mandar el resto de la fbf a evaluar-bicond
Si el primer elemento es =>
Manda a evaluar lo que devuelva el mandar el resto de la fbf a evaluar-cond
Si el primer elemento es un conector n-ario o un literal
Devuelve una lista con ese conector/literal y lo que devuelva evaluar el resto de las expresiones.
En otro caso devuelve la fbf tal cual.

CÓDIGO:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; evaluar
;;; Recibe una expresion y la evalua
;;;
;;; INPUT  : fbf - Formula bien formada (FBF) a analizar
;;; OUTPUT : fbf - Lista la formula bien formada con solo conectores
;;;               n-arios.
;;;
```

```
(defun evaluar (fbf)
  (cond ((literal-p fbf)
         fbf)
    ((unary-connector-p (first fbf))  ;; !(expresion)
      (evaluar (evaluar-not (second fbf))))
    ((bicond-connector-p (first fbf))          ;; Bicond
      (evaluar (evaluar-bicond (rest fbf))))
    ((cond-connector-p (first fbf))           ;; Cond
      (evaluar (evaluar-cond (rest fbf))))
    ((or (literal-p (first fbf)) (n-ary-connector-p (first fbf)))
     (cons (first fbf) (mapcar #'(lambda(x) (evaluar x)) (rest fbf))))
    (t
     fbf))
  )
```

COMENTARIOS:
Va evaluando los elementos de una fórmula bien formada y a través de mandarlos a las diferentes funciones descritas anteriormente consigue devolver una fbf que únicamente contiene conectores n-arios y negaciones.

Apartado 2:

Para este apartado hemos definido 4 funciones además de la pedida por el enunciado (truth-tree). A continuación, detallamos la utilidad de estas y el código de las mismas:

- *CONTRADICCION-LITERALES*

PSEUDOCODIGO:

Entrada: x y literales
Salida: T si se contradicen nil en otro caso

Procesamiento:

Si el primer literal es positivo
Si el segundo literal es también positivo
Devuelve nil
En otro caso devuelve lo que de vuelva el comparar el elemento atómico de ambas.
Si el primer literal es negativo
Si el segundo literal es también negativo
Devuelve nil
En otro caso devuelve lo que de vuelva el comparar el elemento atómico de ambas.

CÓDIGO:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; contradiccion-literales
;;;
;;; Recibe 2 elementos de tipo literal y determina si se contradicen
;;; o no.
;;;
;;; INPUT  : x - Literal
;;;          y - literal
;;; OUTPUT : T -  Si se contradicen
;;;          nil - Si no se contradicen
;;;

(defun contradiccion-literales (x y)
  (if (positive-literal-p x)
      (if (positive-literal-p y)
        nil
        (equal x (second y)))
      (if (negative-literal-p y)
        nil
        (equal (second x) y))
   ))
```

COMENTARIOS:
La razón por la cual evaluamos de esta formas es porque si dos literales son ambos positivos o negativos es imposible que generen una contadicción.

- *CONTRADICCION*

PSEUDOCODIGO:

Entrada: fbf lista de elementos a comparar de una rama
Salida: T si no hay contradicción nil en otro caso

Procesamiento:

Si el resto de lo que recibe es null
Devuelve T
En otro caso si encuentra alguna contradicción al pasar pares a contradicción-literales
Devuelve nil
En otro caso devuelve T

CÓDIGO:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; contradiccion
;;;
;;; Recibe una lista con los literales de una rama y los va comparando
;;; haciendo uso de la función contradiccion-literales y si encuentra
;;; alguna contradiccion devuelve nil, en otro caso devuelve T
;;;
;;; INPUT  : fbf - Lista de atomos a analizar
;;; OUTPUT : T   - Si no encuentra una contradiccion
;;;          N   - Si encuentra alguna contradiccion
;;;

(defun contradiccion (fbf)
   (if (null (rest fbf))
      T
     (if (find T (mapcar #'(lambda (x) (contradiccion-literales x (first fbf))) (rest fbf)))
       nil
       (contradiccion (rest fbf))))
 )
```

COMENTARIOS:
Buscamos alguna contradicción generada en contradicción-literales, si no la hay devolvemos true.

- *COMPARAR*

PSEUDOCODIGO:

Entrada: fbf lista de las listas de cada una de las rama
Salida: T si es SAT nil en otro caso

Procesamiento:

Si es un literal
Si el resto de lo que recibe es null
Devuelve T
En otro caso llama a contradicción de fbf
En otro caso hace la or de llamar a comprar del primer elemento con lo que devuelve comparar del resto

CÓDIGO:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; comparar
;;;
;;; Recibe la lista de las listas de cada una de las rama y a partir
;;; de mandar cada rama a contradiccion, decide si el arbol es SAT o
;;; UNSAT
;;;
;;; INPUT  : fbf - Lista de listas de cada rama
;;; OUTPUT : T   - FBF es SAT
;;;          N   - FBF es UNSAT
;;;

(defun comparar (fbf)
  (if (literal-p (first fbf))
      (if (null (rest fbf))
        T
        (contradiccion fbf))
    (or (comparar (first fbf)) (comparar (rest fbf)))))
```

COMENTARIOS:
Vamos mandado cada una de las listas de cada rama a contradicción y hacemos la or ya que basta que una rama no tenga contradicciones para que la fbf sea SAT

- *TRUTH-TREE-AUX*

PSEUDOCODIGO:

Entrada: fbf lista de las listas de cada una de las rama
Salida: T si es SAT nil en otro caso

Procesamiento:
Si está formada únicamente por una and (^)
Devolvemos la lista de esa rama
Si es un literal
Si la lista de esa rama es null
Devolvemos una lista con ese literal
En otro caso añadimos a la lista de ramas el literal
Si el primer elemento es una or (V)
Hacemos recursión con cada uno de los elementos del resto de la expresión usando mapcar para que genere una lista por cada rama.
Si el primer elemento es una adn (^)
Si el segundo elemento es null
Devolvemos la lista de ramas
En otro caso hacemos recursión mandando como lista de ramas la recursión del segundo elemento y como fbf el resto de la fbf precedida por una and (^)
En otro caso devolvemos la fbf tal cual.

CÓDIGO:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; truth-tree-aux
;;;
;;; Recibe una expresion y construye su arbol de verdad dividiendolo en
;;; ramas las cuales devuelve en forma de lista para ser analizadas.
;;;
;;; INPUT  : fbf - Formula bien formada (FBF) a analizar.
;;;          ramas - lista de ramas
;;; OUTPUT : lista - Lista que contiene cada lista de cada rama.
;;;

(defun truth-tree-aux (ramas fbf)
  (cond ((eql +and+ fbf)
         ramas)
        ((literal-p fbf)
          (if (null ramas)
            (list fbf)
            (append ramas (list fbf))))
        ((eql +or+ (first fbf))
         (mapcar #'(lambda(x) (truth-tree-aux ramas x)) (rest fbf)))
        ((eql +and+ (first fbf))
         (if (null (second fbf))
           ramas
           (truth-tree-aux (truth-tree-aux ramas (second fbf)) (append (list +and+) (cddr fbf)))))
        (t
         fbf)
   ))
```

COMENTARIOS:
Buscamos generar una lista con los literales que forman cada rama.

- *TRUTH-TREE*

PSEUDOCODIGO:

Entrada: fbf lista de las listas de cada una de las rama
Salida: T si es SAT nil si es UNSAT

Procesamiento:
Si es null la fbf
Devuelve nil
En otro caso devuelve lo que devuelva comparar de truth-tree-aux de una lista vacia y evaluar fbf

CÓDIGO:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; truth-tree
;;; Recibe una expresion y construye su arbol de verdad para
;;; determinar si es SAT o UNSAT
;;;
;;; INPUT  : fbf - Formula bien formada (FBF) a analizar.
;;; OUTPUT : T   - FBF es SAT
;;;          N   - FBF es UNSAT
;;;
```

```
(defun truth-tree (fbf)
  (if (null fbf)
    nil
    (comparar (truth-tree-aux '() (evaluar fbf))))
  )
```

COMENTARIOS:
Llama al resto de funciones definidas para que estas determinen si es SAT o UNSAT. No comprobaos que fbf sea NIL o vacia ya que esto se hace durante el proceso.

Pregunta 1: *si en lugar de (^ A (V B C)) tuviésemos (^ A (¬ A) (V B C)), ¿qué sucedería?*

Que sería UNSAT ya que ambas ramas de la fbf contienen A y !A

Pregunta 2: *¿Y en el caso de (^ A (V B C) (¬ A))?*

Que sería UNSAT ya que ambas ramas de la fbf contienen A y !A al igual que en el apartado anterior porque la and (^) conmuta.

Pregunta 3: *estudia la salida del trace mostrada más arriba. ¿Qué devuelve la función expand-truth- tree?*

Devuelve una lista con los elementos literales de cada rama.

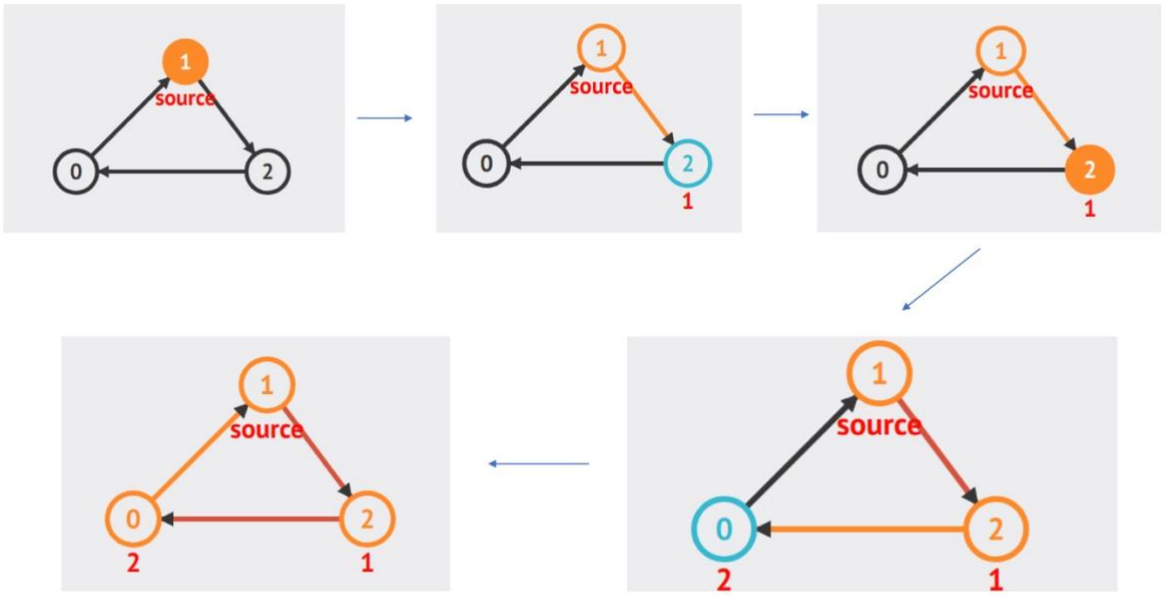
ILUSTRAMOS EL FUNCIONAMIENTO DE LAS FUNCIONES DE ESTE EJERCICIO CON UNA SERIE DE CAPTURAS DE EJEMPLOS QUE SE ENCUENTRAN AL FINAL DEL DOCUMENTO.

Ejercicio 5:

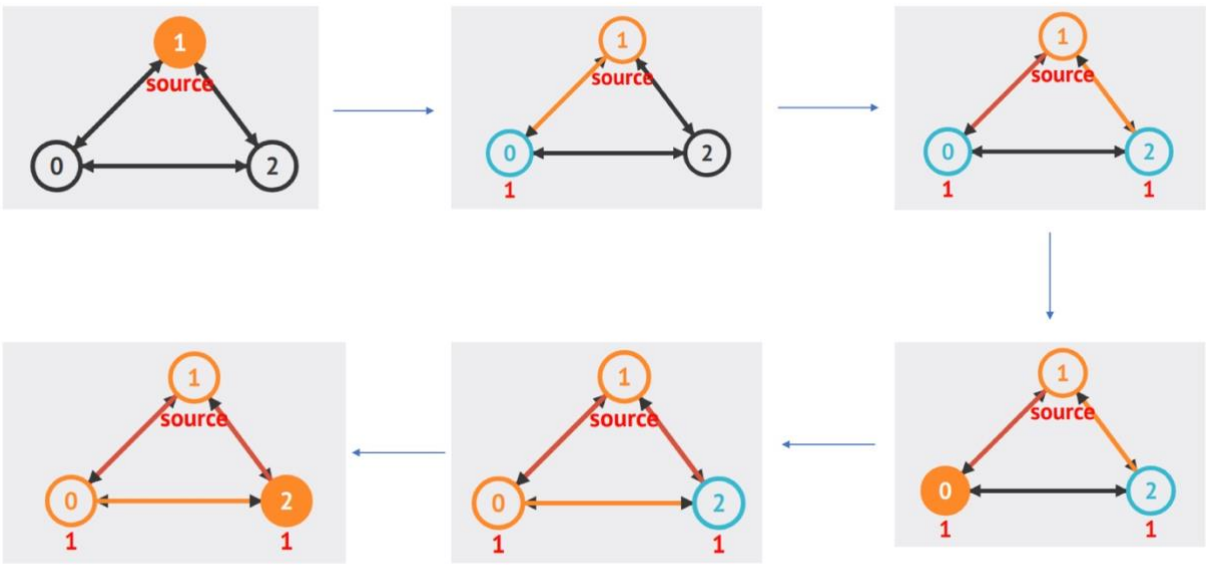
Apartado 1:

Ilustra el funcionamiento del algoritmo resolviendo a mano algunos ejemplos ilustrativos:

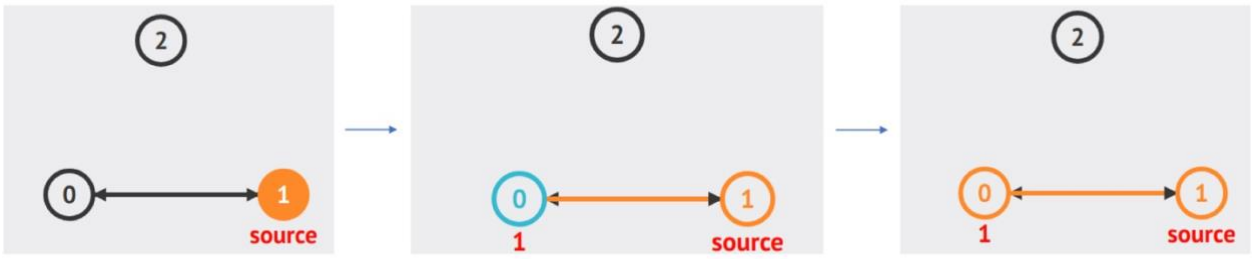
Ejemplo 1:



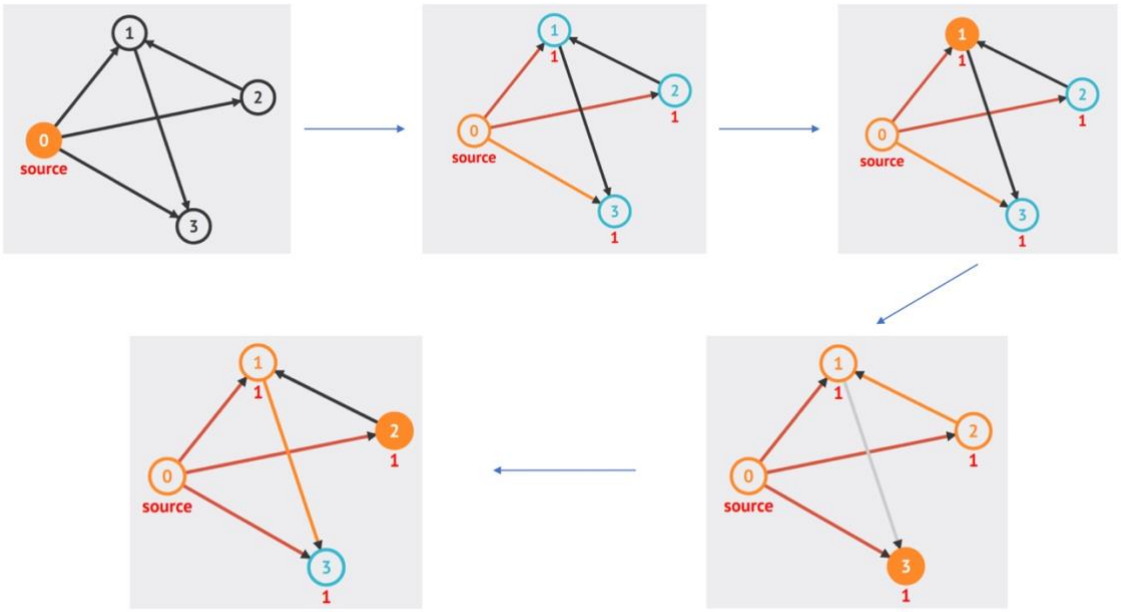
Ejemplo 2:



Ejemplo 3:



Ejemplo 4:



Apartado 2:

Pseudocódigo del bfs:

```
BFS (G, s)                   //Where G is the graph and s is the source node
      let Q be queue
      Q.enqueue( s ) //Inserting s in queue until all its neighbour vertices are marked

      mark s as visited.
      while (Q is not empty)
            //Removing that vertex from queue,whose neighbour will be visited now
            v  =  Q.dequeue( )

            //processing all the neighbours of v
            for all neighbours w of v in Graph G
                  if w is not visited
                        Q.enqueue( w )          //Stores w in Q to further visit its neighbor
                        mark w as visited.
```

Apartado 4:

- *NEW-PATHS*

PSEUDOCODIGO:

Entrada: path, node y net siendo estos el camino seguido hasta ese momento, el nodo a explorar y el grafo

Salida: Todos los posibles caminos desde el nodo

Procesamiento:

Saca todos los nodos vecinos del nodo inicial y los combina con la lista acutal para tener todas las posibles trayectorias

CÓDIGO:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; new-paths
;;; Actualiza los caminos nuevos con los nodos vecinos
;;; INPUT:   path: camino actual
;;;          node: nodo a explorar
;;;          net: grafo
;;; OUTPUT: Todos los posibles caminos desde el nodo

(defun new-paths (path node net)
  (mapcar #'(lambda (n) (cons n path)) (rest (assoc node net)))
  )
```

COMENTARIOS:

Funcion utilizada por bfs y bfs-improved

- *BFS*

PSEUDOCODIGO:

Entrada: end, queue y net siendo estos el nodo final, lista de todos los caminos ya recorridos y el grafo.
Salida: Camino más corto entre dos nodos, nil si no es posible

Procesamiento:
Si el nodo en el que está es el destino invierte el camino seguido y lo devuelve ya que este es el más corto, si no el camino se actualiza con todos los vecinos del nodo actual y se vuelve a intentar.

CÓDIGO:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; bfs
;;; Busqueda en anchura
;;; INPUT:   end: nodo final
;;;          queue: cola de nodos por explorar
;;;          net: grafo
;;; OUTPUT: camino mas corto entre dos nodos
;;;         nil si no lo encuentra

(defun bfs (end queue net)
  (if (null queue) '()
    (let* ((path (first queue))
           (node (first path)))
    (if (eql node end)
      (reverse path)
      (bfs end (append (rest queue) (new-paths path node net))net))))
 )
```

COMENTARIOS:
Funcion utilizada por shortest-path

Apartado 5:

Explica por qué esta función resuelve el problema de encontrar el camino más corto entre dos nodos del grafo:

```
(defun shortest-path (start end net)
  (bfs end (list (list start)) net))
```

Porque al utilizar bfs y no tener peso en los enlaces te garantiza que el camino con menos enlaces es el más corto y ese camino es el primer resultado del algoritmo porque no da preferencia a ningún camino así que en el peor de los casos habrá más de un camino de la misma longitud.

Apartado 6:

Ilustra el funcionamiento del código especificando la secuencia de llamadas a las que da lugar la evaluación:

(shortest-path 'a 'f '((a d) (b d f) (c e) (d f) (e b f) (f)))

```
CL-USER> (shortest-path 'a 'f '(( a d ) ( b d f ) ( c e ) ( d f ) ( e b f ) ( f )))
  0: (BFS F ((A)) ((A D) (B D F) (C E) (D F) (E B F) (F)))
    1: (NEW-PATHS (A) A ((A D) (B D F) (C E) (D F) (E B F) (F)))
    1: NEW-PATHS returned ((D A))
    1: (BFS F ((D A)) ((A D) (B D F) (C E) (D F) (E B F) (F)))
      2: (NEW-PATHS (D A) D ((A D) (B D F) (C E) (D F) (E B F) (F)))
      2: NEW-PATHS returned ((F D A))
      2: (BFS F ((F D A)) ((A D) (B D F) (C E) (D F) (E B F) (F)))
      2: BFS returned (A D F)
    1: BFS returned (A D F)
  0: BFS returned (A D F)
(A D F)
```

Apartado 7:

Utiliza el código anterior para encontrar el camino más corto entre los nodos B y G en el siguiente grafo no dirigido. ¿Cuál es la llamada concreta que tienes que hacer? ¿Qué resultado obtienes con dicha llamada?:

(shortest-path 'b 'g '((a b c e d) (b a d e f) (c a g) (d a b g h) (e a b g h) (f b h) (g c d e h) (h d e f g)))
Hemos tenido que actualizar los enlaces por no ser un grafo dirigido así que los enlaces son bidireccionales.

Con esta llamada se obtiene de resultado (B D G) que es uno de los dos caminos más cortos en este grafo, siendo el otro (B E G) por tanto funciona correctamente.

Apartado 8:

El código anterior falla (entra en una recursión infinita) cuando hay ciclos en el grafo y el problema de búsqueda no tiene solución. Ilustra con un ejemplo este caso problemático y modifica el código para corregir este problema:

(shortest-path 'a 'f '((a d) (b d f) (c e) (d f) (e b) (f e)))
Solamente con el cambio de f→e en vez de e→f se crea un bucle interno en el grafo anterior y al poner como destino c sale un nodo inalcanzable y entra en bucle infinito.

Las funciones actualizadas son las siguientes:

- *BFS-IMPROVED*

PSEUDOCODIGO:

Entrada: end, queue y net siendo estos el nodo final, lista de todos los caminos ya recorridos y el grafo.
Salida: Camino más corto entre dos nodos, nil si no es posible

Procesamiento:
Si el nodo en el que está es el destino invierte el camino seguido y lo devuelve ya que este es el más corto, si no el camino se actualiza con todos los vecinos del nodo actual y se vuelve a intentar, si detecta que el nodo a explorar está ya en el path (se forma un bucle) devuelve nil y termina esa rama.

CÓDIGO:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; bfs-improved
;;; Version de busqueda en anchura que no entra en recursion
;;; infinita cuando el grafo tiene ciclos
;;; INPUT:   end: nodo final
;;;          queue: cola de nodos por explorar
;;;          net: grafo
;;; OUTPUT: camino mas corto entre dos nodos
;;;         nil si no lo encuentra

(defun bfs-improved (end queue net)
  (if (null queue) '() ;; Si no quedan elementos en la cola devuelve una lista vacia para cuando haga
append
    (let* ((path (first queue))
          (node (first path)))
    (if (eql node end)
      (reverse path) ;; Si el nodo actual es el final se invierte el camino seguido para llegar hasta ahi
y se devuelve
      (if (null (find node (rest path)))
        (bfs-improved end (append (rest queue) (new-paths path node net))net)
        nil))))
  )
```

COMENTARIOS:
Funcion utilizada por shortest-path-improved.

- *SHORT-PATH-IMPROVED*

PSEUDOCODIGO:

Entrada: start, end y net siendo estos el nodo incial, el nodo final y el grafo.
Salida: Camino más corto entre dos nodos, nil si no es posible

Procesamiento:
Si el nodo en el que está es el destino invierte el camino seguido y lo devuelve ya que este es el más corto, si no el camino se actualiza con todos los vecinos del nodo actual y se vuelve a intentar, si detecta que el nodo a explorar está ya en el path (se forma un bucle) devuelve nil y termina esa rama.

CÓDIGO:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; shortest-path-improved
;;; Version de busqueda en anchura que no entra en recursion
;;; infinita cuando el grafo tiene ciclos
;;; INPUT:   start: nodo inicial
;;;          end: nodo final
;;;          net: grafo
;;; OUTPUT: camino mas corto entre dos nodos
;;;         nil si no lo encuentra

(defun shortest-path-improved (start end net)
  (bfs-improved end (list (list start)) net)
 )
```

COMENTARIOS:
Esta versión de short-path evita los bucles infinitos.

Capturas Ejercicio 4:

Grupo de funciones convierten una fbf en otra con solo conecotres n-arios y negaciones:

```
CL-USER> (evaluar '(<=> (v A B) C))
 (^ (V (^ (! A) (! B)) C) (V (! C) (V A B)))

CL-USER> (evaluar '(=> (v A (! B)) (^ C D)))
 (V (^ (! A) (B)) (^ C D))

CL-USER> (evaluar '(^ (v A (!B) (=> C D) (<=> (! E) (^ F G))) H))
 (^ (V A (!B) (V (! C) D) (^ (V (E) (^ F G)) (V (V (! F) (! G)) (! E)))) H)
```

Ejemplos de truth-tree sin trace:

```
CL-USER> (truth-tree '(^ (v A B)))
 T

CL-USER> (truth-tree '(^ (! B) B))
 NIL

CL-USER> (truth-tree '(<=> (=> (^ P Q) R) (=> P (v (! Q) R))))
 T
```

Ejemplos truth-tree con trace de truth-tree-aux:

```
CL-USER> (truth-tree '(=> A (^ B (! A))))
  0: (TRUTH-TREE-AUX NIL (V (! A) (^ B (! A))))
    1: (TRUTH-TREE-AUX NIL (! A))
    1: TRUTH-TREE-AUX returned ((! A))
    1: (TRUTH-TREE-AUX NIL (^ B (! A)))
      2: (TRUTH-TREE-AUX NIL B)
      2: TRUTH-TREE-AUX returned (B)
      2: (TRUTH-TREE-AUX (B) (^ (! A)))
        3: (TRUTH-TREE-AUX (B) (! A))
        3: TRUTH-TREE-AUX returned (B (! A))
        3: (TRUTH-TREE-AUX (B (! A)) (^))
        3: TRUTH-TREE-AUX returned (B (! A))
      2: TRUTH-TREE-AUX returned (B (! A))
    1: TRUTH-TREE-AUX returned (B (! A))
  0: TRUTH-TREE-AUX returned (((! A)) (B (! A)))
 T
```

```
CL-USER> (truth-tree '(^ A (! B) (v (! A) C)))
  0: (TRUTH-TREE-AUX NIL (^ A (! B) (V (! A) C)))
    1: (TRUTH-TREE-AUX NIL A)
    1: TRUTH-TREE-AUX returned (A)
    1: (TRUTH-TREE-AUX (A) (^ (! B) (V (! A) C)))
      2: (TRUTH-TREE-AUX (A) (! B))
      2: TRUTH-TREE-AUX returned (A (! B))
      2: (TRUTH-TREE-AUX (A (! B)) (^ (V (! A) C)))
        3: (TRUTH-TREE-AUX (A (! B)) (V (! A) C))
          4: (TRUTH-TREE-AUX (A (! B)) (! A))
          4: TRUTH-TREE-AUX returned (A (! B) (! A))
          4: (TRUTH-TREE-AUX (A (! B)) C)
          4: TRUTH-TREE-AUX returned (A (! B) C)
        3: TRUTH-TREE-AUX returned ((A (! B) (! A)) (A (! B) C))
        3: (TRUTH-TREE-AUX ((A (! B) (! A)) (A (! B) C)) (^))
        3: TRUTH-TREE-AUX returned ((A (! B) (! A)) (A (! B) C))
      2: TRUTH-TREE-AUX returned ((A (! B) (! A)) (A (! B) C))
    1: TRUTH-TREE-AUX returned ((A (! B) (! A)) (A (! B) C))
  0: TRUTH-TREE-AUX returned ((A (! B) (! A)) (A (! B) C))
T
```