Compresion de Secuencias

Francisco Javier Serrano Arrese 180487

Table of Contents

memoria

Compresion de Secuencias es un programa desarrollado con el lenguaje de programacion Ciao Prolog. Este lenguaje de programacion forma parte del paradigma de la programacion logica el cual se estudia en la asignatura de Programacion Declarativa: Logica y restricciones de la ETSIINF UPM.

El proyecto consiste en la contruccion de un programa que permita comprimir secuencias de caracteres en secuencias que definan de forma mas compacta las mismas. As, la secuencia aaaaaaa se comprime en a7. La secuencia original es de longitud siete, la comprimida tiene solo longitud dos. La secuencia ababab, de longitud seis, se comprime en (ab)3, de longitud cinco. Ntese que los parntesis se cuentan tambin como caracteres de la secuencia. Solo hacen falta parntesis si la subsecuencia que se repite es de ms de un carcter. Para comprimir secuencias complejas, se hace una division de tal manera que se pueda comprimir las partes divididas para luego juntarlas en el resultado final de la compresion. Por ejemplo, la secuencia aaaaaaa se comprime en la secuencia a7. En la compresin por divisin se divide la secuencia original en dos partes que a su vez se comprimen por separado y se unen los resultados. Por ejemplo, la secuencia aaaaaaabbbbbbb se comprime en a7b7 (comprimiendo cada parte, a su vez, por repeticin). La secuencia aaabaaab se comprime por repeticin en (a3b)2 donde la subsecuencia aaab se ha comprimido en a3b por divisin (y a su vez aaa en a3 por repeticin). Los resultados de la compresin (tanto de la secuencia original como de sus subsecuencias) han de ser ms cortos que las secuencias iniciales. As, no es admisible comprimir aa en a2, porque tienen la misma longitud, ni abab en (ab)2, porque esta ltima es ms larga. Las secuencias se representaran como listas de caracteres. Por ejemplo, aaa es [a,a,a] y (ab)3 es [(,a,b,),3]. En estas listas los nmeros ocupan una nica posicin, tengan el numero de dgitos que tengan. As, a12 es la lista [a,12] y tiene longitud dos (no tres).

A continuacion, se detalla el codigo y los predicados utilizados para la resolucion de el enunciado anteriormente expuesto junto con un set de pruebas para asegurar el correcto funcionamiento del codigo.

Predicados y explicacion de algoritmos

partir(Todo, Parte1, Parte2)

```
:- pred partir(Todo, Parte1, Parte2)

Se verifica si @var{Parte1} y @var{Parte2} son dos subsecuencias no va-
cias que concatenadas forman la secuencia @var{Todo}.@includedef{partir/3}
```

Explicacion partir(Todo, Parte1, Parte2)

```
El algoritmo utilizado para este predicado cuenta con los sigu-
ientes pasos:
  1) Comprobacion de que @var{Parte1} no es una lista vacia.
  2) Comprobacion de que @var{Parte2} no es una lista vacia.
  3) Utilizar el predicado @var{append/2} para encontrar las lis-
tas que concatenadas formen la lista @var{Todo}.
```

parentesis(Parte, Num, ParteNum)

```
:- pred parentesis(Parte, Num, ParteNum)

Compone la lista de elementos @var{Parte} con el numero de repeti-
ciones dados por la variable @var{Num} envolviendo a esta lista con par-
entesis solo si esta tiene 2 elementos o mas. @includedef{parentesis/3}
```

Explicacion parentesis(Parte, Num, ParteNum)

```
El algoritmo utilizado para este predicado cuenta con los sigu-
ientes pasos:
  1) Se checkea que @var{Num} sea efectivamente un numero entero positivo.
  2) Se comprueba el size de la lista @var{Parte} y se bifurca en-
tre size = 1 o size > 1.
  3) Si @var{Parte} es de size 1, se hace un corte para no en-
volverla en parentesis y se concatena el valor de @var{Parte} con el nu-
mero de @var{Num} en @var{ParteNum}.
  4) Si @var{Parte} es de size > 1, se concatenan parentesis por de-
lante y por detras de la lista en @var{ParteNum}.
```

se_repite(Cs, Parte, Num0, Num)

```
:- pred se_repite(Cs, Parte, Num0, Num)

Se verifica si @var{Cs} se obtiene por repetir N veces la secuen-
cia @var{Parte}. El argumento @var{Num} incre-
menta @var{Num0} en N. @includedef{se_repite/4}
```

Explicacion se_repite(Cs, Parte, Num0, Num)

```
El algoritmo utilizado para este predicado cuenta con los sigu-
ientes pasos:
  1) Se obtiene en @var{X} la lista que concate-
nada a @var{Parte} forma la lista completa @var{Cs}.
  2) Se incrementa en @var{Num1} el valor de @var{Num0}.
  3) Se llama recursivamente a se_repite/4 con las vari-
ables @var{X}, @var{Parte}, @var{Num1}, @var{Num}.
  4) Cuando @var{Cs} esta vacia, se iguala los val-
ores de @var{Num0} y @var{Num}.
```

repeticion(Inicial, Comprimida)

```
:- pred repeticion(Inicial, Comprimida)

Se basa en los predicados anteriormente detalla-
dos @var{partir/3} y @var{se_repite/4}. Este predicado identifica un pre-
fijo que nos de por repeticion la secuencia inicial.
```

Se comprime de forma recursiva mediante un llamada a @var{compresion_recursiva/2}. Finalmente se debe componer la parte (comprimida recursivamente) con el numero de repeticiones usando el predicado @var{parentesis/3}. @includedef{repeticion/2}

Explicacion repeticion(Inicial, Comprimida)

El algoritmo utilizado para este predicado cuenta con los siguientes pasos:
1) Se llama al predicado @var{partir/2} obteniendo asi una seccion de @var{Inicial} en @var{Parte1}.
2) Se llama a @var{se_repite/4} para identificar si la seccion @var{Parte1} se repite en el conjunto de @var{Inicial} y de ser asi, cuantas veces lo hace en @var{Num}.
3) Se llama a @var{compresion_recursiva/2} (posteriormente se hace una llamada a un predicado auxiliar por motivos de eficiencia), obteniendo en @var{X} la secuencia @var{Parte1} comprimida.
4) Por ultimo se hace una llamada a @var{parentesis/3} para envolver a la secuencia @var{X} con su numero de repeticiones @var{Num} y devolverlo en @var{Comprimida}.

divison(Inicial, Comprimida)

:- pred division(Inicial, Comprimida)

Verifica que la lista @var{Inicial} queda dividida en dos partes y llama a @var{compresion_recursiva/2} de forma recursiva para posteriormente concatenar los resultados obtenidos. @includedef{division/2}

Explicacion division(Inicial, Comprimida)

El algoritmo utilizado para este predicado cuenta con los siguientes pasos:
1) Se llama a @var{partir/2} de tal forma que se divide @var{Inicial} en las sublistas @var{X} e @var{Y}.
2) Se llama a @var{compresion_recursiva/2} obteniendo en @var{X1} la compresion de @var{X}.
3) Se llama a @var{compresion_recursiva/2} obteniendo en @var{Y1} la compresion de @var{Y}.
4) Por ultimo, se concatenan las listas @var{X1} e @var{Y1} en la lista @var{Comprimida}.

compresion(Inicial, Comprimida)

:- pred compresion(Inicial, Comprimida)

```
Llama alternativamente a los predica-
dos @var{repeticion/2} y a @var{division/2} detallados anteriormente.
Esto implica que ademas de considerar las repeticiones, podremos di-
vidir la lista inicial en dos partes y aplicar el algo-
ritmo a cada una de ellas por separado, de esto modo consigu-
iendo mas posibilidades de encontrar una repeti-
cion. @includedef{compresion/2}
```

Explicacion compresion(Inicial, Comprimida)

```
El algoritmo utilizado para este predicado cuenta con los sigu-
ientes pasos:
  1) Si la lista @var{Inicial} cuenta con un solo ele-
mento, @var{Comprimida} pasa a ser este elemento.
  2) Se llama a @var{division/2} con los parametros de en-
trada de este predicado.
  3) Por otro lado se llama a @var{repeticion/2} con los mismos paramet-
ros de entrada de este predicado.
```

mejor_compresion(Inicial, Comprimida)

```
:- pred mejor_compresion(Inicial, Comprimida)

Verifica la busqueda de las compresiones que reduzcan el tamao. Deno-
tar que en casos de que no sea posible la compresion, se obten-
dra en @var{Compresion} la lista de en-
trada @var{Inicial}. Se hace uso del predicado de agrega-
cion @var{findall/3} obteniendo todas las soluciones para postenir-
mente filtrar la solucion mas corta. @includedef{mejor_compresion/2}.
```

Explicacion mejor_compresion(Inicial, Comprimida)

```
El algoritmo utilizado para este predicado cuenta con los sigu-
ientes pasos:
  1) Se usa el predi-
cado @var{findall/3} desde el que se llama a @var{get_all_compresions/2} para obtene
das las compresiones posibles en @var{Comp_List}.
  2) Se hace uso de @var{sort/2} para odenar las comrpe-
siones de @var{Comp_List} en funcion del tamao de sus elementos.
  3) Por ultimo hacemos uso del predicado auxil-
iar @var{head/2} para obtener el primer ele-
mento de @var{Sorted_List} obte-
niendo asi la lista con menor tamao en @var{Comprimido}.
```

mejor_compresion_memo(Inicial, Comprimida)

```
:- pred mejor_compresion_memo(Inicial, Comprimida)
```

```
Verifica la busqueda de las compresiones que reduzcan el tamao implemen-
tado un esquema de memorizacion de lemas. Denotar que en ca-
sos de que no sea posible la compresion, se obten-
dra en @var{Compresion} la lista de en-
trada @var{Inicial}. Se hace uso del predicado de agrega-
cion @var{findall/3} obteniendo todas las soluciones para posterior-
mente filtrar la solucion mas corta. @includedef{mejor_compresion_memo/2}
```

Explicacion mejor_compresion_memo(Inicial, Comprimida)

```
El algoritmo utilizado para este predicado cuenta con los sigu-
ientes pasos:
  1) Se llama a @var{limpia_memo/0} para borrar los lemas del predi-
cado @var{memo/2}
  1) Se llama a @var{mejor_compresion/2} para encon-
trar la mejor compresion.
  2) Se hace un corte para no obtener mas compresiones. De no intro-
ducir este corte se obtendia una lista de compresiones ordenadas en fun-
cion de su tamao, lo que es intere-
sante pero no es lo que se pide. @includedef{mejor_compresion_memo}
  NOTA: es importante denotar que el predi-
cado @var{mejor_compresion/2} ya hace uso de la memorizacion de lemas gra-
cias a @var{get_all_compresions/2} por lo que este proceso sera detal-
lado posteriormente en la seccion de predicados auxiliares.
```

Predicados Auxiliares

get_all_compresions(Inicial, Comprimido)

```
:- pred get_all_compresions(Inicial, Comprimido)

Verifica la busqueda de compre-
siones de la lista @var{Inicial} y guarda de forma dinamica los resulta-
dos encontrados. En caso de que una compre-
sion ya haya sido obtenida anteriormente,
esta se descarta. Sim embargo, si estamos ante una nueva compre-
sion, este se guardara dinamicamente como lema del predi-
cado @var{memo/1}. Se ha optado por guardar la compresion en for-
mato Key, Value siendo Key el tamao de la lista y value la lista en si. @includedef{g
```

Explicacion get_all_compresions(Inicial, Comprimida)

```
El algoritmo utilizado para este predicado cuenta con los sigu-
ientes pasos:
  1) Se llama a @var{sub_compresion_recursiva/2} obteniendo una compre-
sion en @var{New_Comp}.
  2) Si @var{New_Comp} ya ha sido encontrada previa-
mente se failea la busqueda de esta compresion en concreto.
```

```
  3) Si @var{New_Comp} no ha sido encontrada previa-
mente, se guarda en @var{memo/1} haciendo uso del predi-
cado @var{store_result/1}
  4) Se haya el tamao de la lista @var{New_Comp} en @var{CL} ha-
ciendo uso del predicado @var{length/2}.
  5) Se verifica que la variable de en-
trada @var{Comprimido} sea igual al par @var{CL} - @var{New_Comp}.
```

store_result(X)

```
:- pred store_result(X)

Guarda de forma dinamica @var{X} como lema del predi-
cado @var{memo/1}.@includedef{store_result/1}
```

Explicacion store_result(X)

```
El algoritmo utilizado para este predicado cuenta con los sigu-
ientes pasos:
  1) Hacer uso de @var{assert/1} para almacenar dinamica-
mente el lema @var{X} en el predicado @var{memo/1}.
```

limpia_memo

```
:- pred limpia_memo

Elimina la memoria dinamica asiganada a los lemas del predi-
cado @var{memo/1}.@includedef{limpia_memo/0}
```

Explicacion limpia_memo

```
El algoritmo utilizado para este predicado cuenta con los sigu-
ientes pasos:
  1) Hacer uso de @var{retractall/1} para eliminar la memoria asi-
ganada los lemas del predicado @var{memo/1}.
```

head(List, Head)

```
:- pred head(List, Head)

Obtiene en @var{Head} el primer elemento de una lista no va-
cia @var{List}.@includedef{head/2}
```

Explicacion head(List, Head)

```
El algoritmo utilizado para este predicado cuenta con los sigu-
ientes pasos:
  1) Establecer una igualdad entre el primer el-
emeto de @var{Lista} y @var{Head}.
```

Casos de prueba

partir(Todo, Parte1, Parte2)

```
Se comprueba caso erroneo en el que Todo es una lista vacia

:- test par-
tir(Todo, Parte1, Parte2) : (Todo=[]) + fails #''No se puede par-
tir una lista vacia''.

Se comprueba caso erroneo en el que Todo es una lista de un elemento

:- test par-
tir(Todo, Parte1, Parte2) : (Todo=[a]) + fails #''No se puede par-
tir una lista con un solo elemento''.

Se comprueba caso valido en el que Todo es una lista con dos elemen-
tos (solo debe dar una solucion).

:- test par-
tir(Todo, Parte1, Parte2) : (Todo=[a,b]) + not_fails #''Lista divi-
dida satisfactoriamente''.

Se comprueba caso valido en el que Todo es una lista con 4 elemen-
tos (debe dar todas las soluciones).

:- test par-
tir(Todo, Parte1, Parte2) : (Todo=[a,b,c,d]) + not_fails #''Lista divi-
dida satisfactoriamente''.
```

parentesis(Parte, Num, ParteNum)

```
Se comprueba caso erroneo en el que Parte no es una lista.

:- test parente-
sis(Parte, Num, ParteNum) : (Parte=1,Num=1) + fails #''No se puede aplicar parentesis

Se comprueba caso erroneo en el que Num no sea un numero.

:- test parente-
sis(Parte, Num, ParteNum) : (Parte=[a,b,c],Num=a) + fails #''No se puede aplicar pare
```

```
Se comprueba caso valido para el primer ejemplo del enunciado.

:- test parente-
sis(Parte, Num, ParteNum) : (Parte=[a,b,c],Num=3) + not_fails #''Parente-
sis aplicado satisfactoriamente''.

Se comprueba caso valido para el segundo ejemplo del enunciado.

:- test parente-
sis(Parte, Num, ParteNum) : (Parte=[a,b],Num=2) + not_fails #''Parente-
sis aplicado satisfactoriamente''.

Se comprueba caso valido para el tercer ejemplo del enunciado.

:- test parente-
sis(Parte, Num, ParteNum) : (Parte=[a],Num=2) + not_fails #''Parente-
sis aplicado satisfactoriamente''.
```

se_repite(Cs, Parte, Num0, Num)

```
Se comprueba caso erroneo en el que Cs no es una lista.

:-
test se_repite(Cs, Parte, Num0, Num) : (Cs=1,Parte=[a,b],Num0=0) + fails #''No se puc
tar el numero de repeticiones de la subsecuencia''.

Se comprueba caso erroneo en el que Parte no es una lista.

:-
test se_repite(Cs, Parte, Num0, Num) : (Cs=[a,b],Parte=1,Num0=0) + fails #''No se puc
tar el numero de repeticiones de la subsecuencia''.

Se comprueba caso erroneo en el que Parte no es subsecuencia de Cs.

:-
test se_repite(Cs, Parte, Num0, Num) : (Cs=[a,b,a,b],Parte=[c,d],Num0=0) + fails #''N
tar el numero de repeticiones de la subsecuencia''.

Se comprueba caso erroneo en el que Parte no es la unica subsecuen-
cia de Cs.

:-
test se_repite(Cs, Parte, Num0, Num) : (Cs=[a,b,a,b,c],Parte=[a,b],Num0=0) + fails #
tar el numero de repeticiones de la subsecuencia''.

Se comprueba caso valido en el que se reconoce una secuencia simple.
```

```
:-
test se_repite(Cs, Parte, Num0, Num) : (Cs=[a,a],Parte=[a],Num0=0) + not_fails #''Se-
cuencia reconocida satisfactoriamente''.

Se comprueba caso valido en el que se reconoce una secuencia de var-
ios caracteres.

:-
test se_repite(Cs, Parte, Num0, Num) : (Cs=[a,b,a,b],Parte=[a,b],Num0=0) + not_fails
cuencia reconocida satisfactoriamente''.

Se comprueba caso valido en el que se reconoce una secuen-
cia con para Num0 distinto de 0.

:-
test se_repite(Cs, Parte, Num0, Num) : (Cs=[a,b,a,b],Parte=[a,b],Num0=5) + not_fails
cuencia reconocida satisfactoriamente''.
```

repeticion(Inicial, Comprimida)

```
Se comprueba caso erroneo en el que no hay repeticiones.

:- test repeticion(Inicial, Comprimida) : (Ini-
cial=[a,b,c]) + fails #''No se pudo comprimir la lista por repeticion''.

Se comprueba caso valido en el que se repite una secuencia de pa-
trones de 3 caracteres.

:- test repeticion(Inicial, Comprimida) : (Ini-
cial=[a,b,c,a,b,c,a,b,c]) + not_fails #''Lista comprimida por repeti-
cion satisfactoriamente''.

Se comprueba caso valido en el que se repite una secuen-
cia del mismo caracter.

:- test repeticion(Inicial, Comprimida) : (Ini-
cial=[a,a,a,a,a,a]) + not_fails #''Lista comprimida por repeti-
cion satisfactoriamente''.

Se comprueba caso valido en el que se repite una secuencia de pa-
trones de 3 caracteres.

:- test repeticion(Inicial, Comprimida) : (Ini-
cial=[a,b,c,a,b,c,a,b,c]) + not_fails #''Lista comprimida por repeti-
cion satisfactoriamente''.
```

compresion(Inicial, Comprimida)

Junto con las comprobaciones de compresion se puede testear el correcto funcionamiento de division.

Se comprueba caso valido en el que no se puede comprimir por lo que se devuelve la misma secuencia.

```
:- test repeticion(Inicial, Comprimida) : (Ini-
cial=[a,b,c]) + not_fails #''Secuencia comprimida satisfactoria-
mente (es la misma que la de entrada)''.
```

Se comprueba caso valido en el que se comprime una secuencia de 3 caracteres 2 veces.

```
:- test repeticion(Inicial, Comprimida) : (Ini-
cial=[a,b,c,a,b,c]) + not_fails #''Secuencia comprim-
ida satisfactoriamente''.
```

Se comprueba caso valido en el que se combina repeticion y division.

```
:- test repeticion(Inicial, Comprimida) : (Ini-
cial=[a,b,a,b,a,b,c,c,c]) + not_fails #''Secuencia comprimida satisfacto-
riamente (es la misma que la de entrada)''.
```

comprimir(Inicial, Comprimido)

Se prueba un caso valido en el que no se puede comprimir por lo que se devuelve la misma secuencia.

```
:- test comprimir(Inicial, Comprimido) : (Ini-
cial=[a,b,c]) + not_fails #''Se ha obtenido la secuencia comprim-
ida optima''.
```

Se prueba un caso valido en el que se comprimen secuencias repetidas de 3 caracateres.

```
:- test comprimir(Inicial, Comprimido) : (Ini-
cial=[a,b,c,a,b,c]) + not_fails #''Se ha obtenido la secuencia comprim-
ida optima''.
```

Se prueba un caso valido en el que se comprime por repeticion un solo elemento varias veces.

```
:- test comprimir(Inicial, Comprimido) : (Ini-
cial=[a,a,a,a,a]) + not_fails #''Se ha obtenido la secuencia comprim-
ida optima''.
```

Se prueba un caso valido en el que se comprime por repeticion y por division varios elementos.

```
:- test comprimir(Inicial, Comprimido) : (Ini-
cial=[a,b,a,b,a,a,a]) + not_fails #''Se ha obtenido la secuencia comprim-
ida optima''.
```

mejor_compresion_memo(Inicial, Comprimido)

```
Se prueba un caso valido en el que no se puede comprimir por lo que se de-
vuelve la misma secuencia.

:- test mejor_compresion_memo(Inicial, Comprimido) : (Ini-
cial=[a,b,c]) + not_fails #''Se ha obtenido la secuencia comprim-
ida optima''.

Se prueba un caso valido en el que se comprimen secuencias repeti-
das de 3 caracateres.

:- test mejor_compresion_memo(Inicial, Comprimido) : (Ini-
cial=[a,b,c,a,b,c]) + not_fails #''Se ha obtenido la secuencia comprim-
ida optima''.

Se prueba un caso valido en el que se comprime por repeticion un solo ele-
mento varias veces.

:- test mejor_compresion_memo(Inicial, Comprimido) : (Ini-
cial=[a,a,a,a,a]) + not_fails #''Se ha obtenido la secuencia comprim-
ida optima''.

Se prueba un caso valido en el que se comprime por repeticion y por divi-
sion varios elementos.

:- test mejor_compresion_memo(Inicial, Comprimido) : (Ini-
cial=[a,b,a,b,a,a,a]) + not_fails #''Se ha obtenido la secuencia comprim-
ida optima''.
```

Usage and interface

- **Library usage:**
  `:- use_module(/home/franser/Documents/uni/3o/Prolog/2-Proyect/memoria/memoria.pl).`
- **Exports:**
  - *Predicates:*
    `alumno_prode/4`, `limpia_memo/0`, `store_result/1`, `compresion_recursiva/2`, `mejor_compresion_memo/2`, `comprimir/2`, `mejor_compresion/2`, `get_all_compresions/2`, `sub_compresion_recursiva/2`, `partir/3`, `parentesis/3`, `se_repite/4`, `repeticion/2`, `compresion/2`, `division/2`, `head/2`.
  - *Multifiles:*
    `Σcall_in_module/2`.

Documentation on exports

alumno_prode/4: PREDICATE

No further documentation available for this predicate.

memo/1: PREDICATE

No further documentation available for this predicate. The predicate is of type *dynamic*.

limpia_memo/0: PREDICATE

Usage:

Elimina los lemas del predicado memo, que son las compresiones encontradas para un secuencia dada como input.

```
limpia_memo :-
    retractall(memo(_1)).
```

store_result/1: PREDICATE

Usage: `store_result(X)`

Guarda de forma dinamica `X` como lema del predicado `memo/1`.

```
store_result(X) :-
    assert(memo(X)).
```

compresion_recursiva/2: PREDICATE

Usage: `compresion_recursiva(Inicial,Comprimido)`

Comprime las listas y recibe llamadas de forma recursiva.

```
compresion_recursiva(Inicial,Comprimido) :-
    limpia_memo,
    mejor_compresion(Inicial,Comprimido).
```

mejor_compresion_memo/2: PREDICATE

Usage: `mejor_compresion_memo(Inicial,Comprimido)`

Verifica la busqueda de las compresiones que reduzcan el tamao implementado un esquema de memorizacion de lemas. Denotar que en casos de que no sea posible la compresion, se obtendra en `Compresion` la lista de entrada `Inicial`. Se hace uso del predicado de agregacion `findall/3` obteniendo todas las soluciones para posteriormente filtrar la solucion mas corta.

```
mejor_compresion_memo(Inicial,Comprimido) :-
    limpia_memo,
    mejor_compresion(Inicial,Comprimido),
    !.
```

comprimir/2: PREDICATE

Usage: `comprimir(Inicial,Comprimido)`

Compresion auxiliar.

```
comprimir(Inicial,Comprimido) :-
    limpia_memo,
    mejor_compresion(Inicial,Comprimido).
```

mejor_compresion/2: PREDICATE

Usage: `mejor_compresion(Inicial,Comprimido)`

Verifica la busqueda de las compresiones que reduzcan el tamao. Denotar que en casos de que no sea posible la compresion, se obtendra en `Compresion` la lista de entrada `Inicial`. Se hace uso del predicado de agregacion `findall/3` obteniendo todas las soluciones para postenirmente filtrar la solucion mas corta.

```
mejor_compresion(Inicial,Comprimido) :-
    findall(Y,get_all_compresions(Inicial,Y),Comp_List),
    sort(Comp_List,Sorted_List),
    head(Sorted_List,[_1-Comprimido]).
```

get_all_compresions/2: PREDICATE

Usage: `get_all_compresions(Inicial,Comprimido)`

Verifica la busqueda de compresiones de la lista `Inicial` y guarda de forma dinamica los resultados encontrados. En caso de que una compresion ya haya sido obtenida anteriormente, esta se descarta. Sim embargo, si estamos ante una nueva compresion, este se guardara dinamicamente como lema del predicado `memo/1`. Se ha optado por guardar la compresion en formato Key, Value siendo Key el tamao de la lista y value la lista en si.

```
get_all_compresions(Inicial,Comprimido) :-
    sub_compresion_recursiva(Inicial,New_Comp),
    ( memo(New_Comp) ->
        fail
    ; store_result(New_Comp),
      length(New_Comp,CL),
      Comprimido=[CL-New_Comp]
    ).
```

sub_compresion_recursiva/2: PREDICATE

Usage: `sub_compresion_recursiva(Inicial,Comprimido)`

Obtiene una compresion unicamente

```
sub_compresion_recursiva(Inicial,Comprimido) :-
    compresion(Inicial,Comprimido).
sub_compresion_recursiva(Inicial,Inicial).
```

partir/3: PREDICATE

Usage: `partir(Todo,Parte1,Parte2)`

Se verifica si `Parte1` y `Parte2` son dos subsecuencias no vacias que concatenadas forman la secuencia `Todo`.

```
partir(Todo,Parte1,Parte2) :-
    Parte1=[_1|_2],
    Parte2=[_3|_4],
    append(Parte1,Parte2,Todo).
```

parentesis/3: PREDICATE

Usage: `parentesis(Parte,Num,ParteNum)`

Compone la lista de elementos `Parte` con el numero de repeticiones dados por la variable `Num` envolviendo a esta lista con parentesis solo si esta tiene 2 elementos o mas.

```
parentesis(Parte,Num,ParteNum) :-
    number(Num),
    length(Parte,1),
    !,
    append(Parte,[Num],ParteNum).
parentesis(Parte,Num,ParteNum) :-
    number(Num),
    append(['('],Parte,ParteAux),
    append(ParteAux,[')',Num],ParteNum).
```

se_repite/4: PREDICATE

Usage: `se_repite([],Arg2,Num0,Num0)`

Se verifica si `Cs` se obtiene por repetir N veces la secuencia `Parte`. El argumento `Num` incrementa `Num0` en N.

```
se_repite([],_1,Num0,Num0).
se_repite(Cs,Parte,Num0,Num) :-
    append(Parte,X,Cs),
    Num1 is Num0+1,
    se_repite(X,Parte,Num1,Num).
```

repeticion/2: PREDICATE

Usage: `repeticion(Inicial,Comprimida)`

Se basa en los predicados anteriormente detallados `partir/3` y `se_repite/4`. Este predicado identifica un prefijo que nos de por repeticion la secuencia inicial. Se comprime de

forma recursiva mediante un llamada a `compresion_recursiva/2`. Finalmente se debe componer la parte (comprimida recursivamente) con el numero de repeticiones usando el predicado `parentesis/3`.

```
repeticion(Inicial,Comprimida) :-
    partir(Inicial,Parte1,_1),
    se_repite(Inicial,Parte1,0,Num),
    sub_compresion_recursiva(Parte1,X),
    parentesis(X,Num,Comprimida).
```

compresion/2: PREDICATE

Usage: `compresion([X],[X])`

Llama alternativamente a los predicados `repeticion/2` y a `division/2` detallados anteriormente. Esto implica que ademas de considerar las repeticiones, podremos dividir la lista inicial en dos partes y aplicar el algoritmo a cada una de ellas por separado, de esto modo consiguiendo mas posibilidades de encontrar una repeticion.

```
compresion([X],[X]) :- !.
compresion(Inicial,Comprimida) :-
    division(Inicial,Comprimida).
compresion(Inicial,Comprimida) :-
    repeticion(Inicial,Comprimida).
```

division/2: PREDICATE

Usage: `division(Inicial,Comprimida)`

Verifica que la lista `Inicial` queda dividida en dos partes y llama a `compresion_recursiva/2` de forma recursiva para posteriormente concatenar los resultados obtenidos.

```
division(Inicial,Comprimida) :-
    partir(Inicial,X,Y),
    sub_compresion_recursiva(X,X1),
    sub_compresion_recursiva(Y,Y1),
    append(X1,Y1,Comprimida).
```

head/2: PREDICATE

Usage: `head([H|_39238],H)`

Obtiene en `Head` el primer elemento de una lista no vacia `List`.

```
head([H|_1],H).
```

Documentation on multifiles

Σcall_in_module/2: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

Documentation on imports

This module has the following direct dependencies:

– *Application modules:*

`operators`, `dcg_phrase_rt`, `datafacts_rt`, `dynamic_rt`, `classic_predicates`, `iso_misc`, `lists`, `sort`.

– *Internal (engine) modules:*

`term_basic`, `arithmetic`, `atomic_basic`, `basiccontrol`, `exceptions`, `term_compare`, `term_typing`, `debugger_support`, `hiord_rt`, `stream_basic`, `io_basic`, `runtime_control`, `basic_props`.

– *Packages:*

`prelude`, `initial`, `condcomp`, `classic`, `runtime_ops`, `dcg`, `dcg/dcg_phrase`, `dynamic`, `datafacts`, `assertions`, `assertions/assertions_basic`.

References

(this section is empty)