mas *Generate & Test*

básicamente programas que generan soluciones idatas que se evalúan para comprobar si son o no ctas

algunas ocasiones es más sencillo comprobar si algo es una ución a un problema que crear la solución a dicho problema

isten en dividir la resolución de problemas en dos s:

erar soluciones candidatas

tear que las soluciones sean correctas

rogramas con la siguiente estructura:

serie de objetivos generan posibles soluciones vía *backtracking*

os objetivos comprueban si dichas soluciones son las apropiadas

mas *Generate & Test*: Ejemplo 1

nación de listas

enacion(X,Y)

es la lista resultante de ordenar la lista X de forma ascendente

a lista Y contiene, en orden ascendente, los mismos elementos que la lista

a lista Y es una permutación de la lista X con los elementos en orden scendente

ordenacion([2,1,2,3], L).

= [1,2,2,3]

enacion(X,Y) :-

ermutacion(X,Y), (1) **Generador**: se obtiene una permutación de X en Y que pasa al objetivo (2) para comprobar si Y está ordenada

rdenada_ascendente(Y). (2) **Prueba**: comprueba si la lista está ordenada. Si no lo está, el *backtracking* se encarga de re-satisfacer el objetivo (1) buscando una nueva permutación

mas *Generate & Test*: Ejemplo 2

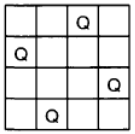
oblema de las N reinas: colocar N reinas en un ro de ajedrez de manera que las reinas no se uen unas a otras

puede haber dos reinas en la misma línea (horizontal, vertical o gonal)

as(N,Tablero)

as(N,Tablero) es cierto si Tablero es una solución al problema de las N reinas.

soluciones se representan como una permutación de la lista de números entre 1 y N: el primer elemento es la n la que se sitúa la reina de la primera columna, el segundo la fila de la reina de la segunda columna, y así sivamente.



[2,4,1,3]

mas *Generate & Test*: Ejemplo 2

```
as(N,Tablero) :-
ange(1,N,L),                    Crea la lista de números entre 1 y N
ermutation(L,Tablero),          Crea una permutación de la lista
afe(Tablero).                   Comprueba si la permutación es solución al problema

e([]).
e([Q|Qs]) :- safe(Qs), not (attack(Q,Qs)).
ck(X,Xs) :- attack(X,1,Xs).
ck(X,N,[Y|Ys]) :- X is Y+N; X is Y-N.
ck(X,N,[Y|Ys]) :- N1 is N+1, attack(X,N1,Ys).
```

a solución es ineficiente:

e generan muchas permutaciones que no pueden ser solución

n solución más eficiente consiste en aplicar la **técnica de los muladores**

ca de los Acumuladores (I)

plo: Inversa de una lista

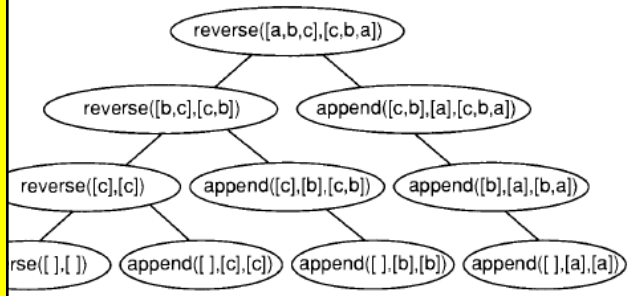
erse(Xs,Ys): Ys es la lista que se obtiene al invertir los elementos a lista Xs

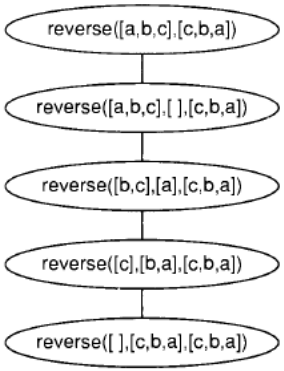
```
everse([],[]).
everse([X|Xs],Ys) :- reverse(Xs,Zs), append(Zs,[X],Ys).
```

reverse/2 es ineficiente

```
everse(Xs,Ys) :- reverse(Xs,[],Ys).
everse([],Ys,Ys).
everse([X|Xs],Acc,Ys) :- reverse(Xs,[X|Acc],Ys).
```

Otra opción (reverse/2) usando **acumuladores**





ca de los Acumuladores (II)

nuladores:

argumentos adicionales usados en los predicados para acenar resultados intermedios

e usan para simular algoritmos iterativos

variables lógicas y su valor se pasa entre iteraciones

plo 1: Factorial

```
orial(N,F) :- factorial(0,N,1,F).

orial(N,N,F,F).

orial(I,N,T,F) :-
< N,
1 is I+1,
1 is T*I1,
actorial(I1,N,T1,F).
```

Este programa lógico simula el comportamiento de un programa iterativo con bucle *while* (de 0 a N)

El **primer argumento** en factorial/4 es el contador del bucle

El **tercer argumento** en factorial/4 es el acumulador de los productos calculados

ca de los Acumuladores (III)

plo 2: Otra versión de Factorial

```
orial(N,F) :- factorial(N,1,F).

orial(0,F,F).

orial(N,T,F) :-
 > 0,
1 is T*N,
1 is N-1,
actorial(N1,T1,F).
```

Versión iterativa de factorial desde N hasta 0 (bucle *while*)

El **segundo argumento** en factorial/3 actúa como acumulador de los productos calculados

rsión 2 es más eficiente que la versión 1

malmente, cuantos menos argumentos tiene un predicado, más
do y más entendible es

cio: *Generate & Test*

ir el predicado numeroParMenor/2 que es
adero cuando X es un numero par menor que N

umeroParMenor(X,5).
=0; X=2; X=4

umeroParMenor(2,4).
es

umeroParMenor(3,5).
o

umeroParMenor(10,7).
o

cio: *Generate & Test*

tegia: generar todos los números entre 0 y N y robar si son pares

```
nParMenor(X,N) :-
ntre(0,N,X),
ar(X).

(X) :- 0 is X mod 2.
```

re es el encargado de generar todos los números menores que N
actúa como filtro