Tarea: Recursividad vs Lógica

Objetivo

Implementar **30 ejercicios** en dos paquetes:

- `com.docencia.recursividad`: solución **recursiva** (y backtracking donde aplique).
- `com.docencia.logica`: solución **iterativa / tradicional** (bucles, estructuras de datos, DP, etc.).

Cada ejercicio está en una clase `Ejercicio01` ... `Ejercicio30` y expone una función `public static` con nombre en español.

> **Regla:** En el paquete `recursividad` **NO** usar bucles para resolver (salvo validaciones o utilidades mínimas). En el paquete `logica` **NO** usar recursividad.

Cómo ejecutar

```
mvn clean test
```

Lista de ejercicios (firma)

1. `factorial(int n) : int/long`
2. `potencia(long base, int exponente) : long`
3. `sumaHastaN(int n) : long`
4. `fibonacci(int n) : long`
5. `mcd(int a, int b) : int`
6. `cuentaRegresiva(int n) : List<Integer>`
7. `sumaDigitos(int n) : int`
8. `contarDigitos(int n) : int`
9. `invertirNumero(int n) : int`
10. `invertirCadena(String texto) : String`
11. `sumaArreglo(int[] arreglo) : long`
12. `maximoArreglo(int[] arreglo) : int`
13. `buscarLineal(int[] arreglo, int objetivo) : int`
14. `esPalindromo(String texto) : boolean`
15. `contarCaracter(String texto, char c) : int`
16. `eliminarCaracter(String texto, char c) : String`
17. `estaOrdenadoAsc(int[] arreglo) : boolean`
18. `fusionarOrdenados(int[] a, int[] b) : int[]`
19. `busquedaBinaria(int[] arregloOrdenado, int objetivo) : int`
20. `mergeSort(int[] arreglo) : int[]`
21. `quickSort(int[] arreglo) : int[]`
22. `potenciaRapida(long base, int exponente) : long`
23. `contarInversiones(int[] arreglo) : long`
24. `encontrarPico(int[] arregloMontana) : int`

25. `generarPermutaciones(int[] arreglo) : List<List<Integer>>` (recomendado: **únicas**)
26. `generarCombinaciones(int[] arreglo, int k) : List<List<Integer>>`
27. `generarSubconjuntos(int[] arreglo) : List<List<Integer>>`
28. `contarSolucionesNReinas(int n) : int`
29. `existeCaminoLaberinto(int[][] lab, int fi, int ci, int ff, int cf) : boolean` (0 libre, 1 bloqueado)
30. `existeSubconjuntoSuma(int[] arreglo, int objetivo) : boolean`

Ejercicio 01 — `factorial(n)`

En qué consiste: devolver `n!` para `n >= 0`.

Lógica (iterativa) — algoritmo: acumulador `r=1`, multiplicar de `2..n`.

```
factorial(n):
  si n < 0 -> error
  r <- 1
  para i <- 2 hasta n:
    r <- r * i
  devolver r
```

Recursividad — algoritmo: caso base `n<=1`; si no, `n * factorial(n-1)`.

```
factorial(n):
  si n < 0 -> error
  si n <= 1 -> devolver 1
  devolver n * factorial(n-1)
```

Ejercicio 02 — `potencia(base, exponente)`

En qué consiste: calcular `base^exponente` con `exponente >= 0`.

Lógica (iterativa)

```
potencia(base, exp):
  si exp < 0 -> error
  r <- 1
  repetir exp veces:
    r <- r * base
  devolver r
```

Recursividad

```
potencia(base, exp):
  si exp < 0 -> error
  si exp == 0 -> devolver 1
  devolver base * potencia(base, exp-1)
```

Ejercicio 03 — `sumaHastaN(n)`

En qué consiste: sumar `1 + 2 + ... + n` (para `n>=0`).

Lógica (iterativa)

```
sumaHastaN(n):
  si n < 0 -> error
  s <- 0
  para i <- 1 hasta n:
    s <- s + i
  devolver s
```

Recursividad

```
sumaHastaN(n):
  si n < 0 -> error
  si n == 0 -> devolver 0
  devolver n + sumaHastaN(n-1)
```

Ejercicio 04 — `fibonacci(n)`

En qué consiste: devolver `F(n)` con `F(0)=0`, `F(1)=1`.

Lógica (iterativa)

```
fibonacci(n):
  si n < 0 -> error
  si n <= 1 -> devolver n
  a <- 0; b <- 1
  para i <- 2 hasta n:
    c <- a + b
    a <- b
    b <- c
  devolver b
```

Recursividad (simple)

```
fibonacci(n):
  si n < 0 -> error
  si n <= 1 -> devolver n
  devolver fibonacci(n-1) + fibonacci(n-2)
```

Ejercicio 05 — `mcd(a, b)`

En qué consiste: máximo común divisor (Euclides).

Lógica (iterativa)

```
mcd(a,b):
  a <- abs(a); b <- abs(b)
  mientras b != 0:
    t <- a mod b
    a <- b
    b <- t
  devolver a
```

Recursividad

```
mcd(a,b):
  a <- abs(a); b <- abs(b)
  si b == 0 -> devolver a
  devolver mcd(b, a mod b)
```

Ejercicio 06 — `cuentaRegresiva(n)`

En qué consiste: generar lista `[n, n-1, ..., 0]` (si `n>=0`).

Lógica (iterativa)

```
cuentaRegresiva(n):
  si n < 0 -> error
  lista <- []
  para i <- n hasta 0 paso -1:
    añadir i a lista
  devolver lista
```

Recursividad

```
cuentaRegresiva(n):
  si n < 0 -> error
  devolver helper(n)

helper(i):
  si i == 0 -> devolver [0]
  devolver [i] + helper(i-1)
```

Ejercicio 07 — `sumaDigitos(n)`

En qué consiste: sumar los dígitos de `n` (usar `abs`).

Lógica (iterativa)

```
sumaDigitos(n):
  n <- abs(n)
  s <- 0
  mientras n > 0:
    s <- s + (n mod 10)
    n <- n / 10
  devolver s
```

Recursividad

```
sumaDigitos(n):
  n <- abs(n)
  si n < 10 -> devolver n
  devolver (n mod 10) + sumaDigitos(n / 10)
```

Ejercicio 08 — `contarDigitos(n)`

En qué consiste: contar dígitos decimales de `n` (convención: `0` tiene 1 dígito).

Lógica (iterativa)

```
contarDigitos(n):
  n <- abs(n)
  si n == 0 -> devolver 1
  c <- 0
  mientras n > 0:
    c <- c + 1
    n <- n / 10
  devolver c
```

Recursividad

```
contarDigitos(n):
  n <- abs(n)
  si n < 10 -> devolver 1
  devolver 1 + contarDigitos(n / 10)
```

Ejercicio 09 — `invertirNumero(n)`

En qué consiste: invertir los dígitos (ej. `123 -> 321`). Recomendado: conservar signo.

Lógica (iterativa)

```
invertirNumero(n):
  signo <- (n<0 ? -1 : 1)
  n <- abs(n)
  r <- 0
  mientras n > 0:
    r <- r*10 + (n mod 10)
    n <- n / 10
  devolver signo*r
```

Recursividad (con ayuda de longitud)

```
invertirNumero(n):
  signo <- (n<0 ? -1 : 1)
  n <- abs(n)
  devolver signo * inv(n, contarDigitos(n))

inv(n, len):
  si len == 1 -> devolver n
  ultimo <- n mod 10
  devolver ultimo * 10^(len-1) + inv(n/10, len-1)
```

Ejercicio 10 — `invertirCadena(texto)`

En qué consiste: devolver el texto al revés.

Lógica (iterativa)

```
invertirCadena(s):
  r <- ""
  para i <- longitud(s)-1 hasta 0 paso -1:
```

```
    r <- r + s[i]
  devolver r
```

Recursividad

```
invertirCadena(s):
  si s es "" o longitud(s)==1 -> devolver s
  devolver ultimoCaracter(s) + invertirCadena(s sin ultimo)
```

Ejercicio 11 — `sumaArreglo(arreglo)`

En qué consiste: sumar elementos.

Lógica (iterativa)

```
sumaArreglo(a):
  s <- 0
  para cada x en a:
    s <- s + x
  devolver s
```

Recursividad

```
sumaArreglo(a):
  devolver sumaDesde(a, 0)

sumaDesde(a, i):
  si i == longitud(a) -> devolver 0
  devolver a[i] + sumaDesde(a, i+1)
```

Ejercicio 12 — `maximoArreglo(arreglo)`

En qué consiste: máximo elemento (arreglo no vacío).

Lógica (iterativa)

```
maximoArreglo(a):
  max <- a[0]
  para i <- 1 hasta fin:
    si a[i] > max: max <- a[i]
  devolver max
```

Recursividad

```
maximoArreglo(a):
  devolver maxDesde(a, 0)

maxDesde(a, i):
  si i == longitud(a)-1 -> devolver a[i]
  m <- maxDesde(a, i+1)
  devolver max(a[i], m)
```

Ejercicio 13 — `buscarLineal(arreglo, objetivo)`

En qué consiste: devolver índice o `-1`.

Lógica (iterativa)

```
buscarLineal(a, obj):
  para i <- 0 hasta fin:
    si a[i] == obj -> devolver i
  devolver -1
```

Recursividad

```
buscarLineal(a, obj):
  devolver buscarDesde(a, obj, 0)

buscarDesde(a, obj, i):
  si i == longitud(a) -> devolver -1
  si a[i] == obj -> devolver i
  devolver buscarDesde(a, obj, i+1)
```

Ejercicio 14 — `esPalindromo(texto)`

En qué consiste: true si se lee igual al revés (normalmente sin modificar caso/espacios salvo que lo pidas).

Lógica (iterativa)

```
esPalindromo(s):
  i <- 0; j <- longitud(s)-1
  mientras i < j:
    si s[i] != s[j] -> devolver false
```

```
    i <- i+1; j <- j-1
  devolver true
```

Recursividad

```
esPalindromo(s):
  devolver pal(s, 0, longitud(s)-1)

pal(s, i, j):
  si i >= j -> devolver true
  si s[i] != s[j] -> devolver false
  devolver pal(s, i+1, j-1)
```

Ejercicio 15 — `contarCaracter(texto, c)`

En qué consiste: contar ocurrencias del carácter.

Lógica (iterativa)

```
contarCaracter(s, c):
  cnt <- 0
  para cada ch en s:
    si ch == c: cnt <- cnt+1
  devolver cnt
```

Recursividad

```
contarCaracter(s, c):
  si s es "" -> devolver 0
  suma <- (primer(s)==c ? 1 : 0)
  devolver suma + contarCaracter(resto(s), c)
```

Ejercicio 16 — `eliminarCaracter(texto, c)`

En qué consiste: devolver cadena sin ese carácter.

Lógica (iterativa)

```
eliminarCaracter(s, c):
  r <- ""
  para cada ch en s:
```

```
    si ch != c: r <- r + ch
  devolver r
```

Recursividad

```
eliminarCaracter(s, c):
  si s es "" -> devolver ""
  ch <- primer(s)
  si ch == c -> devolver eliminarCaracter(resto(s), c)
  devolver ch + eliminarCaracter(resto(s), c)
```

Ejercicio 17 — `estaOrdenadoAsc(arreglo)`

En qué consiste: true si `a[i] <= a[i+1]` para todo i.

Lógica (iterativa)

```
estaOrdenadoAsc(a):
  para i <- 0 hasta longitud(a)-2:
    si a[i] > a[i+1] -> devolver false
  devolver true
```

Recursividad

```
estaOrdenadoAsc(a):
  devolver ordDesde(a, 0)

ordDesde(a, i):
  si i >= longitud(a)-1 -> devolver true
  si a[i] > a[i+1] -> devolver false
  devolver ordDesde(a, i+1)
```

Ejercicio 18 — `fusionarOrdenados(a, b)`

En qué consiste: merge de dos arrays ordenados asc.

Lógica (iterativa)

```
fusionarOrdenados(a,b):
  i<-0; j<-0; r<-[]
  mientras i<|a| o j<|b|:
    si j==|b| o (i<|a| y a[i] <= b[j]):
      añadir a[i]; i++
```

```
    si no:
      añadir b[j]; j++
  devolver r
```

Recursividad

```
fusionarOrdenados(a,b):
  devolver merge(a,0,b,0)

merge(a,i,b,j):
  si i==|a| -> devolver b[j..fin]
  si j==|b| -> devolver a[i..fin]
  si a[i] <= b[j] -> devolver [a[i]] + merge(a,i+1,b,j)
  devolver [b[j]] + merge(a,i,b,j+1)
```

Ejercicio 19 — `busquedaBinaria(arregloOrdenado, objetivo)`

En qué consiste: índice o `-1` en array ordenado.

Lógica (iterativa)

```
busquedaBinaria(a, obj):
  lo<-0; hi<-|a|-1
  mientras lo <= hi:
    mid <- (lo+hi)/2
    si a[mid]==obj -> devolver mid
    si a[mid]<obj -> lo<-mid+1
    si no -> hi<-mid-1
  devolver -1
```

Recursividad

```
busquedaBinaria(a,obj):
  devolver bin(a,obj,0,|a|-1)

bin(a,obj,lo,hi):
  si lo>hi -> devolver -1
  mid <- (lo+hi)/2
  si a[mid]==obj -> devolver mid
  si a[mid]<obj -> devolver bin(a,obj,mid+1,hi)
  devolver bin(a,obj,lo,mid-1)
```

Ejercicio 20 — `mergeSort(arreglo)`

En qué consiste: ordenar por Merge Sort.

Lógica (iterativa / tradicional)

```
mergeSort(a):
  (versión iterativa típica: bottom-up)
  ancho <- 1
  mientras ancho < |a|:
    para i <- 0 hasta |a|-1 paso 2*ancho:
      fusionar runs [i..i+ancho) y [i+ancho..i+2*ancho)
    ancho <- 2*ancho
  devolver a
```

Recursividad

```
mergeSort(a):
  si |a| <= 1 -> devolver a
  mid <- |a|/2
  izq <- mergeSort(a[0..mid))
  der <- mergeSort(a[mid..fin))
  devolver fusionarOrdenados(izq, der)
```

Ejercicio 21 — `quickSort(arreglo)`

En qué consiste: ordenar por Quick Sort.

Lógica (iterativa)

```
quickSort(a):
  usar pila de rangos (lo,hi)
  push(0,|a|-1)
  mientras pila no vacía:
    (lo,hi) <- pop
    p <- particionar(a, lo, hi)
    push(lo, p-1) y push(p+1, hi) si procede
  devolver a
```

Recursividad

```
quickSort(a):
  qs(a,0,|a|-1)
  devolver a

qs(a,lo,hi):
  si lo>=hi -> return
```

```
  p <- particionar(a,lo,hi)
  qs(a,lo,p-1)
  qs(a,p+1,hi)
```

Ejercicio 22 — `potenciaRapida(base, exponente)`

En qué consiste: exponenciación rápida O(log n), `exp>=0`.

Lógica (iterativa)

```
potenciaRapida(b, e):
  si e < 0 -> error
  r <- 1
  mientras e > 0:
    si (e es impar): r <- r*b
    b <- b*b
    e <- e/2
  devolver r
```

Recursividad

```
potenciaRapida(b, e):
  si e < 0 -> error
  si e == 0 -> devolver 1
  si e es par:
    t <- potenciaRapida(b, e/2)
    devolver t*t
  devolver b * potenciaRapida(b, e-1)
```

Ejercicio 23 — `contarInversiones(arreglo)`

En qué consiste: contar pares `(i<j)` con `a[i] > a[j]`. (Clásico con merge).

Lógica (iterativa)

```
contarInversiones(a):
  inv <- 0
  para i<-0..|a|-1:
    para j<-i+1..|a|-1:
      si a[i] > a[j]: inv++
  devolver inv
```

Recursividad (merge sort + conteo)

```
contarInversiones(a):
  devolver invSort(a).inv

invSort(a):
  si |a|<=1 -> devolver (a,0)
  mid <- |a|/2
  (izq, invL) <- invSort(a[0..mid))
  (der, invR) <- invSort(a[mid..fin))
  (m, invM) <- mergeContando(izq, der)
  devolver (m, invL + invR + invM)

mergeContando(izq,der):
  cuando eliges der[j] antes que izq[i], sumas (|izq|-i)
```

Ejercicio 24 — `encontrarPico(arregloMontana)`

En qué consiste: en un “array montaña” (sube y luego baja), devolver índice del pico.

Lógica (iterativa, binaria)

```
encontrarPico(a):
  lo<-0; hi<-|a|-1
  mientras lo < hi:
    mid <- (lo+hi)/2
    si a[mid] < a[mid+1]:
      lo <- mid+1
    si no:
      hi <- mid
  devolver lo
```

Recursividad

```
encontrarPico(a):
  devolver pico(a,0,|a|-1)

pico(a,lo,hi):
  si lo==hi -> devolver lo
  mid <- (lo+hi)/2
  si a[mid] < a[mid+1] -> devolver pico(a, mid+1, hi)
  devolver pico(a, lo, mid)
```

Ejercicio 25 — `generarPermutaciones(arreglo)`

En qué consiste: listar permutaciones. Recomendado: evitar duplicadas si hay repetidos.

Lógica (iterativa)

```
generarPermutaciones(a):
  ordenar(a)
  lista <- [a]
  mientras nextPermutation(a) exista:
    añadir copia(a) a lista
  devolver lista
```

Recursividad (backtracking)

```
permutar(a):
  ordenar(a)
  usado[|a|] <- false
  actual <- []
  res <- []
  bt():
    si |actual|==|a|: añadir copia(actual) a res; return
    para i<-0..|a|-1:
      si usado[i] continuar
      si i>0 y a[i]==a[i-1] y no usado[i-1] continuar (evita duplicados)
      usado[i]<-true; añadir a[i]
      bt()
      quitar último; usado[i]<-false
  bt()
  devolver res
```

Ejercicio 26 — `generarCombinaciones(arreglo, k)`

En qué consiste: elegir subconjuntos de tamaño `k`.

Lógica (iterativa)

```
combinaciones(a,k):
  usar bitmask / índices crecientes
  idx <- [0,1,...,k-1]
  mientras exista siguiente idx:
    añadir a[idx[*]] a res
  devolver res
```

Recursividad (backtracking)

```
combinaciones(a,k):
  res<-[]; actual<-[]
  bt(pos, restantes):
```

```
    si restantes==0: añadir copia(actual); return
    si pos==|a|: return
    # elegir
    añadir a[pos]; bt(pos+1, restantes-1); quitar último
    # no elegir
    bt(pos+1, restantes)
  bt(0,k)
  devolver res
```

Ejercicio 27 — `generarSubconjuntos(arreglo)`

En qué consiste: power set (todos los subconjuntos).

Lógica (iterativa)

```
subconjuntos(a):
  res <- [[]]
  para cada x en a:
    para cada s en copia(res):
      res.add(s + [x])
  devolver res
```

Recursividad

```
subconjuntos(a):
  res<-[]; actual<-[]
  bt(i):
    si i==|a|: añadir copia(actual); return
    bt(i+1)            # no tomar
    añadir a[i]
    bt(i+1)            # tomar
    quitar último
  bt(0)
  devolver res
```

Ejercicio 28 — `contarSolucionesNReinas(n)`

En qué consiste: contar cuántas formas de colocar `n` reinas sin atacarse.

Lógica (iterativa)

```
NReinas(n):
  (típico: backtracking con pila explícita)
```

```
  usar pila de estados (fila, columnas/diagonales ocupadas)
  iterar simulando DFS y contar cuando fila==n
```

Recursividad (backtracking)

```
contarSolucionesNReinas(n):
  cols[n] <- false
  diag1[2n] <- false   # r-c+n
  diag2[2n] <- false   # r+c
  return dfs(0)

dfs(r):
  si r==n -> devolver 1
  total <- 0
  para c<-0..n-1:
    si cols[c] o diag1[r-c+n] o diag2[r+c] continuar
    marcar; total += dfs(r+1); desmarcar
  devolver total
```

Ejercicio 29 — `existeCaminoLaberinto(lab, fi, ci, ff, cf)`

En qué consiste: devolver si hay camino de inicio a fin (4 direcciones), con `0` libre, `1` bloqueado.

Lógica (iterativa)

```
existeCamino(lab, ini, fin):
  BFS/DFS con cola/pila
  visitado[][] <- false
  push ini
  mientras estructura no vacía:
    (r,c) <- pop
    si (r,c)==fin -> true
    para cada vecino válido libre no visitado:
      marcar y push
  devolver false
```

Recursividad (DFS)

```
existeCamino(lab, r, c, ff, cf):
  si fuera de límites o lab[r][c]==1 -> false
  si visitado[r][c] -> false
  si (r,c)==(ff,cf) -> true
  visitado[r][c] <- true
  return existeCamino(r+1,c) o existeCamino(r-1,c) o existeCamino(r,c+1) o
existeCamino(r,c-1)
```

Ejercicio 30 — `existeSubconjuntoSuma(arreglo, objetivo)`

En qué consiste: decidir si algún subconjunto suma `objetivo`.

Lógica (iterativa / DP)

```
existeSubconjuntoSuma(a, objetivo):
  dp[0..objetivo] <- false
  dp[0] <- true
  para cada x en a:
    para s desde objetivo hasta x:
      dp[s] <- dp[s] o dp[s-x]
  devolver dp[objetivo]
```

Recursividad (backtracking)

```
existeSubconjuntoSuma(a, objetivo):
  return bt(0, objetivo)

bt(i, restante):
  si restante == 0 -> true
  si i == |a| -> false
  # tomar
  si bt(i+1, restante - a[i]) -> true
  # no tomar
  return bt(i+1, restante)
```