Documentación: Recursividad

1. Definición propia de recursividad (para entregar y comentar)

Definición:
La recursividad es una técnica de solución de problemas en programación en la que una función se define en términos de sí misma; es decir, la función resuelve una instancia del problema llamándose a sí misma para resolver subinstancias más pequeñas hasta alcanzar un caso base que detiene las llamadas recursivas. Esta técnica permite dividir un problema complejo en subproblemas de la misma naturaleza y facilita soluciones claras para estructuras y algoritmos que tienen una naturaleza naturalmente jerárquica o repetitiva. Wikipedia

Comentario para trinas / discusión corta:
La recursividad destaca por su claridad conceptual: en muchos problemas (árboles, recorridos, definiciones matemáticas como el factorial) la solución recursiva es más directa y fácil de razonar que la versión iterativa. Sin embargo, hay que controlar el consumo de pila (stack) y asegurar siempre la existencia de una condición de salida que impida bucles infinitos. GeeksforGeeks+1

2. Ventajas y desventajas de la recursividad (para la plenaria)

Ventajas

- **Claridad y expresividad:** Permite escribir soluciones muy legibles para problemas que se definen de forma recursiva (p. ej. recorridos de árboles, backtracking). GeeksforGeeks
- **Divide y vencerás:** Facilita el diseño de algoritmos que dividen el problema en subproblemas más pequeños (p. ej. QuickSort, MergeSort). GeeksforGeeks
- **Menos código en muchos casos:** Puede reducir la complejidad del código y evitar bucles anidados difíciles de manejar. DataCamp

Desventajas

- **Coste en memoria (call stack):** Cada llamada recursiva consume marco de pila; en problemas muy profundos puede producirse *stack overflow*. Stack Overflow
- **Rendimiento:** En ciertas soluciones (p. ej. recursión naïve para Fibonacci) hay recomputación de subproblemas y es menos eficiente que una versión iterativa o que una versión recursiva con memoización. Stack Overflow+1
- **Restricciones del entorno:** No todos los lenguajes o compiladores aplican optimizaciones como *tail-call optimization*, por lo que la recursión profunda puede ser peligrosa. Stack Overflow

3. Trasladar un catálogo de problemas iterativos a recursivos

(Aquí tienes varios ejemplos típicos; para cada uno indico **qué** convertir y **cómo identificar** segmento recursivo y condición de salida.)

Ejemplo A — Factorial (n!)

- **Problema iterativo (concepto):** multiplicar 1·2·…·n usando un bucle `for`.
- **Versión recursiva (estructura):**
  - **Caso base:** n == 0 ó n == 1 → devolver 1.
  - **Segmento recursivo:** devolver `n * factorial(n - 1)`.
- **Comentario:** la función se reduce en cada llamada hasta llegar a 1; es la transformación prototipo para entender recursión. W3Schools

Ejemplo B — Fibonacci (serie)

- **Problema iterativo:** calcular fib(n) con bucle acumulando dos últimos valores.
- **Versión recursiva (estructura):**
  - **Casos base:** n == 0 → 0; n == 1 → 1.
  - **Segmento recursivo:** `fib(n) = fib(n - 1) + fib(n - 2)`.
- **Comentario:** la versión recursiva directa tiene recomputación exponencial; explore memoización o iterativo para eficiencia. GeeksforGeeks

Ejemplo C — Suma de elementos de un arreglo

- **Iterativo:** sumar con un bucle.
- **Recursivo:**
  - **Caso base:** arreglo vacío → 0; o índice == última posición.
  - **Segmento recursivo:** `sum(arr, i) = arr[i] + sum(arr, i+1)`.
- **Comentario:** patrón de "divide por un elemento" que facilita pruebas inductivas.

Ejemplo D — Búsqueda binaria

- **Iterativo:** repetir con limites `low`/`high`.
- **Recursivo:**
  - **Caso base:** `low > high` → no encontrado; o `mid == target` → encontrado.
  - **Segmento recursivo:** llamar a la mitad derecha o izquierda: `binarySearch(arr, low, mid-1, key)` o `binarySearch(arr, mid+1, high, key)`.
- **Comentario:** la definición recursiva refleja la idea de partición por mitades y suele quedar muy clara en forma recursiva. GeeksforGeeks

Ejemplo E — Recorrido de árbol (inorden, preorden, postorden)

- **Caso base:** nodo == null → retornar.
- **Segmento recursivo:** llamar recursivamente a hijo izquierdo/derecho y procesar el nodo según el orden.
- **Comentario:** recursividad es la forma natural de expresar recorridos de estructuras jerárquicas. GeeksforGeeks

4. Plantilla para el reporte de práctica (qué debe contener el documento final)

Usa este formato para cada ejercicio recursivo que entregues:

1. **Título del ejercicio** (p. ej. Factorial recursivo)
2. **Objetivo** (qué se pretende demostrar)
3. **Descripción del problema** (enunciado claro)
4. **Versión iterativa original** (pseudocódigo o explicación breve; tú incluyes el código)

5. **Versión recursiva propuesta** (pseudocódigo o esquema — sin código si prefieres)
6. **Identificación clara:**
    a. **Caso base (condición de salida):** explicar por qué detiene la recursión.
    b. **Segmento recursivo:** mostrar la expresión que llama a la misma función con la subinstancia.
7. **Razonamiento de corrección:** breve argumento inductivo por el que la solución es correcta.
8. **Complejidad:** analizar complejidad temporal y espacial (pe. O(n), O(n) extra por pila, etc.).
9. **Pruebas:** casos de prueba sugeridos (mínimo 3), entradas y salidas esperadas.
10. **Conclusión y observaciones:** problemas de eficiencia, posibles mejoras (memoización, conversión a iterativo, tail recursion).

5. Sugerencia de conjunto de ejercicios (catálogo) para convertir y entregar

- Factorial (iterativo → recursivo)
- Fibonacci (iterativo → recursivo; luego recursivo con memoización)
- Suma de arreglo (recursivo por índice)
- Búsqueda binaria (iterativa → recursiva)
- Inversión de cadena (iterativa → recursiva)
- Torres de Hanoi (ejercicio clásico recursivo)
- Recorridos de árbol: preorden, inorden, postorden (ejemplo con árbol binario)
- Generar permutaciones (backtracking recursivo)

Para cada uno aplica la plantilla de reporte.

6. Recomendaciones para los programas que vas a desarrollar (qué debes incluir en el informe)

- **Comenta tu código** (explica cuál es el caso base y cuál el caso recursivo con comentarios en el código).
- **Incluye mediciones simples** (por ejemplo, tiempo de ejecución para n pequeño/medio/grande) si la práctica lo requiere.

- **Si hay problemas de rendimiento**, explica alternativas (iterativo, memoización, uso de estructuras auxiliares).
- **Menciona límites de pila**: si pruebas con valores grandes, anota si ocurre `StackOverflowException` o equivalente.
- **Adjunta resultados de pruebas**: entradas, salidas y observaciones.
- **Conclusión final**: cuándo conviene usar recursión y cuándo no, en base a tus pruebas.

7. Ejemplo breve de cómo identificar en cualquier ejercicio (resumen práctico)

- Paso 1: Encuentra la versión más simple del problema → ese será el **caso base**.
- Paso 2: Expresa la resolución del problema en términos de la resolución de una subinstancia más simple → esa expresión es el **segmento recursivo**.
- Paso 3: Asegura que cada llamada recursiva acerque la instancia al caso base (decremento o partición).
- Paso 4: Analiza consumo de pila y costo temporal.

8. Bibliografía / fuentes consultadas

- Wikipedia – Definición y ejemplos de recursión https://es.wikipedia.org/wiki/Recursi%C3%B3n
- GeeksforGeeks – Introducción, ventajas y ejemplos prácticos https://www.geeksforgeeks.org/dsa/what-is-recursion/
- W3Schools – Caso base y llamada recursiva (ejemplos) https://www.w3schools.com/python/python_recursion.asp
- Stack Overflow – Ventajas y desventajas de la recursividad https://stackoverflow.com/questions/5250733/what-are-the-advantages-and-disadvantages-of-recursion
- DataCamp – Guía práctica sobre recursividad https://www.datacamp.com/tutorial/recursion-python