

Bloque A — Arrays (10 ejercicios)

A1) Recorrido básico y suma ("recorrer un vector")

Enunciado: Dado un array de enteros `nums`, recórrelo y calcula la **suma** y el **máximo** sin usar `reduce` ni `Math.max`. **Resultado esperado:** Para `[3,7,2,9]` → `suma=21`, `max=9`. **Ayuda parcial:**

```
const nums = [3, 7, 2, 9];
let suma = 0;
let max = -Infinity;

for (* TODO: índice i de 0..len-1 *) {
    const x = /* TODO: elemento actual */;
    suma = /* TODO: acumular */;
    if (* TODO: x mayor que max *) {
        max = x;
    }
}

console.log({ suma, max });
```

A2) Búsqueda secuencial ("buscar en un array")

Enunciado: Implementa `indexOfManual(arr, valor)` devolviendo el **índice** o `-1` si no aparece (recorrido lineal). **Resultado esperado:** `indexOfManual(["a","b","c"], "b") -> 1`. **Ayuda parcial:**

```
function indexOfManual(arr, valor) {
    for (let i = 0; i < arr.length; i++) {
        // TODO: compara arr[i] con valor
    }
    return -1;
}
```

A3) Insertar y eliminar con `splice` ("desplazar elementos")

Enunciado: Dado `cola = ["A", "B", "C"]`, inserta `"X"` en la **posición 1** y elimina el elemento en la **posición 2**. **Resultado esperado:** Tras ambas operaciones → `['A', 'X', 'C']`. **Ayuda parcial:**

```
const cola = ["A", "B", "C"];
cola.splice(* TODO: pos, deleteCount, item *);
cola.splice(* TODO: pos, deleteCount *);
console.log(cola);
```

A4) Clonado vs referencia ("copias de arrays")

Enunciado: Clona `origen=[1,2,3]` en `copia` sin compartir referencia. Modifica `copia[0]=99` y demuestra que `origen` no cambia. **Resultado esperado:** `origen=[1,2,3]`, `copia=[99,2,3]`. **Ayuda parcial:**

```
const origen = [1, 2, 3];
const copia = /* TODO: usa spread o slice */;

copia[0] = 99;
console.log({ origen, copia });
```

A5) Eliminar duplicados preservando orden ("conjuntos/set")

Enunciado: Implementa `unicos(arr)` que devuelva un nuevo array sin repetidos **manteniendo el primer aparecido**. **Resultado esperado:** `[1,2,2,3,1,4] -> [1,2,3,4]`. **Ayuda parcial:**

```
function unicos(arr) {
  const vistos = new Set();
  return arr.filter((x) => {
    // TODO: si no visto, agregar a Set y mantener
  });
}
```

A6) Ordenación por selección ("ordenar un vector")

Enunciado: Implementa `selectionSort(arr)` que ordene **in-place** de menor a mayor **sin** usar `sort`. **Resultado esperado:** `[5,2,4,1] -> [1,2,4,5]`. **Ayuda parcial:**

```
function selectionSort(arr) {
  for (let i = 0; i < arr.length - 1; i++) {
    let minIdx = i;
    for (let j = i + 1; j < arr.length; j++) {
      // TODO: actualizar minIdx si arr[j] < arr[minIdx]
    }
    // TODO: intercambiar arr[i] con arr[minIdx] usando destructuring
  }
  return arr;
}
```

A7) Insertar en array ordenado (búsqueda + desplazamiento)

Enunciado: Dado `asc=[10,20,30,40]` inserta `25` manteniendo el orden ascendente. Implementa `insertaOrdenado(arr, x)`. **Resultado esperado:** `[10,20,25,30,40]`. **Ayuda parcial:**

```

function insertaOrdenado(arr, x) {
  let i = 0;
  while /* TODO: i < arr.length && arr[i] < x */) i++;
  arr.splice(/* TODO: posición i, 0, x */);
  return arr;
}

```

A8) Matriz 2D: generar tabla $n \times n$ ("matrices")

Enunciado: Crea `cuadrado(n)` que devuelva una matriz $n \times n$ con valores $\text{fila} * n + \text{col} + 1$.

Resultado esperado: $n=3 \rightarrow [[1,2,3],[4,5,6],[7,8,9]]$. Ayuda parcial:

```

function cuadrado(n) {
  const m = [];
  for (let i = 0; i < n; i++) {
    const fila = [];
    for (let j = 0; j < n; j++) {
      fila.push(/* TODO: fórmula */);
    }
    m.push(fila);
  }
  return m;
}

```

A9) Suma y traspuesta de matrices

Enunciado: Escribe `sumaM(A,B)` y `traspuesta(M)` para matrices rectangulares válidas. **Resultado**

esperado: `sumaM([[1,2],[3,4]],[[5,6],[7,8]]) -> [[6,8],[10,12]]`

`traspuesta([[1,2,3],[4,5,6]]) -> [[1,4],[2,5],[3,6]]`. Ayuda parcial:

```

function sumaM(A, B) {
  // TODO: comprobar dimensiones y sumar elemento a elemento
}

function traspuesta(M) {
  const filas = M.length;
  const cols = M[0]?.length ?? 0;
  const T = Array.from({ length: cols }, () => Array(filas).fill(0));
  for (let i = 0; i < filas; i++) {
    for (let j = 0; j < cols; j++) {
      T[j][i] = /* TODO: asignar M[i][j] */;
    }
  }
  return T;
}

```

A10) Histograma de notas (map/filter/reduce)

Enunciado: Dadas notas `0..10`, construye un objeto **frecuencias** `{0:_,1:_,...,10:_}` y calcula **media** y **moda**. **Resultado esperado:** Objeto con frecuencias, `media` y `moda` correctos. **Ayuda parcial:**

```
const notas = [4,7,10,7,5,7,6,9,4];
const freq = Object.fromEntries(Array.from({ length: 11 }, (_, k) => [k, 0]));

for (const n of notas) {
    // TODO: incrementar freq[n]
}

const media = /* TODO: suma(notas) / notas.length */;
const moda = /* TODO: clave con mayor frecuencia */;

console.log({ freq, media, moda });
```

Bloque B — Funciones (10 ejercicios)

F1) Declaración, expresión y arrow (hoisting)

Enunciado: Escribe `decl()`, `const expr = function(){};`, y `const arr = () => {}.` Demuestra qué dos **no** se pueden usar antes de su definición y explica por qué. **Resultado esperado:** Llamadas correctas/incorrectas y breve comentario. **Ayuda parcial:**

```
// TODO: prueba llamar a cada una antes y después de definirlas
function decl() { return "ok"; }
const expr = function() { return "ok"; };
const arr = () => "ok";
```

F2) Parámetros por defecto y `rest`

Enunciado: Implementa `precioTotal(base, iva=0.21, ...descuentos)` que reste todos los `descuentos` tras aplicar IVA. **Resultado esperado:** `precioTotal(100, 0.1, 5, 2) -> 103.` **Ayuda parcial:**

```
function precioTotal(base, iva = 0.21, ...descuentos) {
    let total = /* TODO: base * (1 + iva) */;
    for (const d of descuentos) total -= /* TODO */;
    return Number(total.toFixed(2));
}
```

F3) Función pura vs con efectos

Enunciado: Escribe una función **pura** `neto(precio, iva)` y otra **impura** que además haga `console.log`. Explica por qué una es pura y la otra no. **Resultado esperado:** Cálculo idéntico; diferencia conceptual. **Ayuda parcial:**

```
const neto = (precio, iva) => /* TODO: devolver nuevo valor sin tocar nada
externo */;
function netoImpuro(precio, iva) {
  const r = /* TODO */;
  console.log("Total:", r); // efecto colateral
  return r;
}
```

F4) Higher-order: `map`, `filter`, `reduce`

Enunciado: Dado `productos=[{p:10},{p:5},{p:12}]`, obtén (1) precios con IVA 21%, (2) solo los `>10`, y (3) la **suma** total. **Resultado esperado:** Tres resultados coherentes. **Ayuda parcial:**

```
const productos = [{ p: 10 }, { p: 5 }, { p: 12 }];
const conIva = productos.map(/* TODO: devolver {p:..., total:...} */);
const mayores = conIva.filter(/* TODO: total > 10 */);
const suma = mayores.reduce((acc, x) => /* TODO */, 0);
console.log({ conIva, mayores, suma });
```

F5) Closures: contador con límites

Enunciado: `crearContador(min=0, max=Infinity)` devuelve `{inc, dec, get, reset}` que saturan en los límites. **Resultado esperado:** No sobrepasa `[min,max]`. **Ayuda parcial:**

```
function crearContador(min = 0, max = Infinity) {
  let x = min;
  return {
    inc() { /* TODO: x = Math.min(x+1, max) */ },
    dec() { /* TODO: x = Math.max(x-1, min) */ },
    get() { return x; },
    reset() { x = min; },
  };
}
```

F6) Recursión: factorial y validación

Enunciado: Implementa `fact(n)` recursivo que lance Error si `n<0` o no es entero. **Resultado esperado:** `fact(5)->120`, `fact(-1)` lanza. **Ayuda parcial:**

```

function fact(n) {
  if (!Number.isInteger(n) || n < 0) throw new Error("n inválido");
  if (n <= 1) return 1;
  return /* TODO: n * fact(n-1) */;
}

```

F7) compose y pipe

Enunciado: Implementa `compose(...fns)` (derecha→izquierda) y `pipe(...fns)` (izquierda→derecha). Pruébalas con `inc` y `dup`. **Resultado esperado:** Misma entrada, órdenes distintos. **Ayuda parcial:**

```

const compose = (...fns) => (x) => /* TODO: reduceRight */;
const pipe = (...fns) => (x) => /* TODO: reduce */;
const inc = (x) => x + 1;
const dup = (x) => x * 2;
console.log({ compose: compose(dup, inc)(3), pipe: pipe(inc, dup)(3) });

```

F8) Currificación ligera

Enunciado: `multiplicador(a)` devuelve una función que multiplica por `a`. Crea `por2` y `por5` y úsalos con una lista. **Resultado esperado:** `[1,2,3] -> por2 -> [2,4,6]` y `por5 -> [5,10,15]`. **Ayuda parcial:**

```

const multiplicador = (a) => /* TODO: devolver (b) => a * b */;
console.log([1,2,3].map(/* TODO: por2 */));
console.log([1,2,3].map(/* TODO: por5 */));

```

F9) Memoización genérica

Enunciado: Crea `memo(fn)` que guarde resultados por clave de argumentos (usa `Map`). Úsalo para acelerar `fib(n)`. **Resultado esperado:** `fibMemo(40)` sensiblemente más rápido que sin memo. **Ayuda parcial:**

```

function memo(fn) {
  const cache = new Map();
  return (...args) => {
    const key = JSON.stringify(args);
    if (cache.has(key)) return /* TODO: devolver cache */;
    const r = fn(...args);
    cache.set(key, r);
    return r;
  };
}

```

```
const fib = (n) => (n <= 1 ? n : fib(n - 1) + fib(n - 2));
const fibMemo = memo(fib);
```

F10) Validadores combinables (and / or)

Enunciado: Implementa `and(...rules)` y `or(...rules)` donde cada `rule` es `(s)=>boolean`. Crea `minLen(8)`, `hasUpper()`, `hasDigit()` y compón `isStrong`. **Resultado esperado:** `isStrong("Abcdef12") -> true`. **Ayuda parcial:**

```
const minLen = (n) => (s) => s.length >= n;
const hasUpper = () => (s) => /[A-Z]/.test(s);
const hasDigit = () => (s) => /\d/.test(s);

const and = (...rules) => (s) => /* TODO: todas verdaderas */;
const or = (...rules) => (s) => /* TODO: alguna verdadera */;

const isStrong = and(minLen(8), hasUpper(), hasDigit());
```