Desafío final: tu primer proyecto cuántico

El proyecto final reúne el curso en un entregable concreto: un notebook que construye circuitos, los verifica en simulación ideal, mide el efecto del ruido y resuelve un problema con un algoritmo cuántico. No se trata de hacer el circuito más grande posible, sino de demostrar que entiendes qué hace cada parte y cómo interpretar los resultados.

Qué debes entregar

Un notebook que corra de principio a fin sin errores, con estas tres partes:

- **Parte 1. Entrelazamiento:** una función `ghz(n)` que prepare un estado GHZ de *n* qubits. Verifica con `Statevector` que solo tienen probabilidad los estados de todos ceros y todos unos.
- **Parte 2. Efecto del ruido:** ejecuta el GHZ con 2, 3, 4 y 5 qubits en un simulador con ruido (`AerSimulator.from_backend` con un backend falso). Calcula la *fidelidad* de cada caso como la proporción de disparos que caen en los resultados correctos y grafícala contra *n*.
- **Parte 3. Algoritmo:** implementa Bernstein-Vazirani para una cadena secreta de al menos cinco bits. Ejecútalo en ideal y con ruido, y reporta qué porcentaje de disparos recupera el secreto.

Cierra con un texto breve, de tres a cinco frases, que explique qué observaste y por qué.

Estructura sugerida

Separa el notebook en secciones con una celda de texto al inicio de cada una. Escribe funciones reutilizables en lugar de copiar código entre celdas. El siguiente bloque muestra el esqueleto de la Parte 2:

```
from qiskit import QuantumCircuit, transpile
from qiskit_aer import AerSimulator
from qiskit_ibm_runtime.fake_provider import FakeManilaV2

def ghz(n):
    qc = QuantumCircuit(n)
    qc.h(0)
    for i in range(n - 1):
```

```
        qc.cx(i, i + 1)
    qc.measure_all()
    return qc

sim = AerSimulator.from_backend(FakeManilaV2(), seed_simulator=42)
for n in [2, 3, 4, 5]:
    circuito = transpile(ghz(n), sim, seed_transpiler=42)
    conteos = sim.run(circuito, shots=2000).result().get_counts()
    correctos = conteos.get("0" * n, 0) + conteos.get("1" * n, 0)
    fidelidad = correctos / 2000
    print(n, "qubits -> fidelidad:", round(fidelidad, 3))
```

Deberías ver que la fidelidad baja al aumentar *n*. Explicar por qué es parte central del proyecto: más qubits significa más puertas `cx`, y cada una agrega error.

Criterios de evaluación

- **Corrección:** los circuitos ideales producen exactamente los resultados esperados.
- **Uso de las herramientas:** `Statevector` para verificar, primitivas o `AerSimulator` para muestrear y un backend falso para el ruido.
- **Análisis:** la gráfica y el texto final explican la relación entre tamaño del circuito, ruido y fidelidad.
- **Reproducibilidad:** semillas fijas en simuladores y transpilador; el notebook corre completo tras reiniciar el kernel.

Extensiones opcionales

- Ejecuta la Parte 3 en un procesador real de IBM Quantum y compara con la simulación ruidosa.
- Prueba los niveles de optimización 0 a 3 del transpilador y reporta cómo cambian la profundidad y la fidelidad.
- Agrega Grover de dos qubits y verifica que encuentra cada uno de los cuatro estados posibles.

Trampas comunes

- **Comparar sin semillas.** Los resultados cambian en cada ejecución y las conclusiones se vuelven difíciles de sostener.
- **Leer las cadenas al revés.** Revisa el orden de los bits en Bernstein-Vazirani antes de concluir que el algoritmo falló.

- **Olvidar transpilar.** El simulador con ruido necesita el circuito en las puertas nativas del procesador.
- **Conclusiones sin datos.** Cada afirmación del texto final debe apoyarse en un número o una gráfica del notebook.

Cómo autoevaluar antes de entregar

Reinicia el kernel y ejecuta todo. Luego lee solo las gráficas y el texto final: si alguien que no vio el código puede entender qué hiciste y qué aprendiste, el proyecto está listo. El notebook del proyecto incluye una solución de referencia para comparar tu trabajo cuando termines.