Test de Lógica para Desarrollador .NET Junior - Resuelto

SECCIÓN 1: RAZONAMIENTO LÓGICO (25 puntos)

1.1 Secuencias Lógicas

1. 3, 6, 12, 24, **48, 96**
   - *Explicación*: Cada número es el doble del anterior (3×2=6, 6×2=12, 12×2=24, 24×2=48, 48×2=96)

2. 1, 4, 9, 16, 25, **36**
   - *Explicación*: Son los cuadrados de los números naturales ($1=1^2$, $4=2^2$, $9=3^2$, $16=4^2$, $25=5^2$, $36=6^2$)

3. 3, 6, 8, 16, 18, 36, **38, 76**
   - *Explicación*: Se alterna entre "sumar 3 (luego 2, luego 2, etc.)" y "multiplicar por 2"
     - 3 → +3 → 6 → ×2 → 12 → +4 → 16 → ×2 → 32 → +6 → 38 → ×2 → 76

4. A, C, F, J, **O, T**
   - *Explicación*: Cada letra ocupa la posición correspondiente a los números triangulares en el alfabeto
     - A (1ª), C (3ª), F (6ª), J (10ª), O (15ª), T (21ª)

1.2 Silogismos y Deducciones Lógicas

5. Todas las aplicaciones web modernas usan bases de datos. Este software no usa bases de datos. Por lo tanto, este software no es una aplicación web moderna.
   - **Respuesta**: V (Válido)
   - *Explicación*: Es la forma lógica "Todos los A son B. X no es B. Por lo tanto, X no es A." (Modus Tollens)

6. Todos los bugs de software tienen solución. Algunas soluciones requieren reescribir código. Por lo tanto, algunos bugs de software requieren reescribir código.
   - **Respuesta**: V (Válido)
   - *Explicación*: Es la forma lógica "Todos los A son B. Algunos B son C. Por lo tanto, algunos A son C."

7. Ningún lenguaje de programación es perfecto. C# es un lenguaje de programación. Algunos lenguajes de programación son difíciles de aprender. Por lo tanto, C# es difícil de aprender.
   - **Respuesta**: I (Inválido)
   - *Explicación*: No hay conexión lógica entre "C# es un lenguaje de programación" y "algunos lenguajes son difíciles" para concluir que C# pertenece a ese subconjunto.

1.3 Problemas de Pensamiento Lateral

8. Un programador escribe una función que toma un número entero positivo como entrada y devuelve otro número. Cuando la entrada es 1, devuelve 5. Cuando la entrada es 2, devuelve 6. Cuando la entrada es 3, devuelve 7. Cuando la entrada es 4, devuelve 8. ¿Qué valor devuelve cuando la entrada es 10?
   - **Respuesta**: 14
   - *Explicación*: La función simplemente suma 4 al número de entrada. Por tanto, 10 + 4 = 14.
9. Un desarrollador tiene que implementar un algoritmo para agrupar clientes. Si divide a los clientes en grupos de 3, sobran 2. Si los divide en grupos de 5, sobran 3. Si los divide en grupos de 7, sobran 2. ¿Cuál es el mínimo número de clientes que podría tener?
   - **Respuesta**: 23
   - *Explicación*: Necesitamos un número que:
     - Al dividir por 3, dé resto 2 → $n \equiv 2 \pmod 3$
     - Al dividir por 5, dé resto 3 → $n \equiv 3 \pmod 5$
     - Al dividir por 7, dé resto 2 → $n \equiv 2 \pmod 7$
     - Aplicando el teorema chino del resto, el número más pequeño que cumple estas condiciones es 23.
     - Verificación: 23 ÷ 3 = 7 con resto 2; 23 ÷ 5 = 4 con resto 3; 23 ÷ 7 = 3 con resto 2.

SECCIÓN 2: ALGORITMOS Y ESTRUCTURAS DE DATOS (30 puntos)

2.1 Complejidad y Eficiencia

10. Ordene los siguientes algoritmos de ordenamiento de menor a mayor complejidad en el caso promedio: Bubble Sort, Quick Sort, Merge Sort.
    - **Respuesta**: Quick Sort ≈ Merge Sort < Bubble Sort
    - *Explicación*:
      - Bubble Sort: $O(n^2)$ en caso promedio
      - Quick Sort: O(n log n) en caso promedio
      - Merge Sort: O(n log n) en caso promedio
      - Quick Sort y Merge Sort tienen la misma complejidad teórica en el caso promedio, aunque Quick Sort suele ser más rápido en la práctica por constantes ocultas más bajas.
11. ¿Cuál es la complejidad temporal de buscar un elemento en un array no ordenado? ¿Y en un array ordenado usando búsqueda binaria?
    - **Respuesta**:
      - Array no ordenado: O(n) - en el peor caso hay que recorrer todo el array
      - Array ordenado con búsqueda binaria: O(log n) - en cada paso se reduce el espacio de búsqueda a la mitad

2.2 Resolución de Problemas Algorítmicos

12. Describa un algoritmo para encontrar el segundo número más grande en un array no ordenado. Analice su complejidad temporal y espacial.
    - **Respuesta**: Un algoritmo eficiente sería:

```
función segundoMayor(array):
    si longitud(array) < 2:
        retornar "No hay segundo mayor"

    mayor = -∞
    segundoMayor = -∞

    para cada elemento en array:
        si elemento > mayor:
            segundoMayor = mayor
            mayor = elemento
        sino si elemento < mayor Y elemento > segundoMayor:
            segundoMayor = elemento

    si segundoMayor == -∞:
        retornar "Todos los elementos son iguales"
    sino:
        retornar segundoMayor
```

    - Complejidad temporal: O(n) - se recorre el array una sola vez
    - Complejidad espacial: O(1) - solo se utilizan dos variables adicionales

13. Escriba pseudocódigo para verificar si una palabra es un palíndromo (se lee igual de izquierda a derecha que de derecha a izquierda).
    - **Respuesta**:

```
función esPalindromo(palabra):
    inicio = 0
    fin = longitud(palabra) - 1

    mientras inicio < fin:
        si palabra[inicio] != palabra[fin]:
            retornar falso
        inicio = inicio + 1
        fin = fin - 1

    retornar verdadero
```

14. Explique con sus propias palabras cómo implementaría una cola (queue) usando dos pilas (stacks).
    - **Respuesta**: Una implementación eficiente utilizaría dos pilas:
        - Una pila de "entrada" (pila1) donde se añaden los nuevos elementos
        - Una pila de "salida" (pila2) desde donde se extraen los elementos

      Las operaciones serían:
        - **Enqueue(elemento)**: Simplemente hacer push del elemento a pila1
        - **Dequeue()**:
            - Si pila2 está vacía, transferir todos los elementos de pila1 a pila2 (haciendo pop de pila1 y push a pila2)
            - Hacer pop de pila2 y devolver ese elemento

      Este enfoque garantiza que el primer elemento añadido será el primero en salir (FIFO), con una complejidad amortizada de O(1) para cada operación.

2.3 Análisis de Código

15. ¿Cuál es la salida del siguiente código? Explique paso a paso.

```csharp
int[] numbers = { 1, 2, 3, 4, 5 };
int result = 0;
for (int i = 0; i < numbers.Length; i++) {
    if (numbers[i] % 2 == 0) {
        result += numbers[i];
    }
}
Console.WriteLine(result);
```

- **Respuesta**: 6
- *Explicación*:
    - El código suma los números pares del array.
    - numbers[0] = 1: No es par, no se suma.
    - numbers[1] = 2: Es par, result = 0 + 2 = 2.
    - numbers[2] = 3: No es par, no se suma.
    - numbers[3] = 4: Es par, result = 2 + 4 = 6.
    - numbers[4] = 5: No es par, no se suma.
    - Al final, result = 6.

16. ¿Cuál es el problema en el siguiente código? ¿Cómo lo solucionaría?

csharp

```csharp
public List<int> FindDivisors(int n) {
    List<int> divisors = new List<int>();
    for (int i = 1; i <= n; i++) {
        if (n % i == 0) {
            divisors.Add(i);
        }
    }
    return divisors;
}
```

- **Respuesta**: El código es funcionalmente correcto, pero es ineficiente para números grandes. Podemos optimizarlo recorriendo solo hasta la raíz cuadrada de n.
- *Solución*:

csharp

```csharp
public List<int> FindDivisors(int n) {
    List<int> divisors = new List<int>();
    for (int i = 1; i <= Math.Sqrt(n); i++) {
        if (n % i == 0) {
            divisors.Add(i);
            if (i != n / i) { // Evitar duplicados en números perfectos cuadrados
                divisors.Add(n / i);
            }
        }
    }
    divisors.Sort(); // Opcional, si queremos los divisores en orden
    return divisors;
}
```

17. ¿Cuál es la salida del siguiente código? Explique el resultado.

csharp

```csharp
int x = 10;
int y = x++;
int z = ++x;
Console.WriteLine($"x={x}, y={y}, z={z}");
```

- **Respuesta**: x=12, y=10, z=12
- *Explicación*:
  - x++ es un postincremento: primero asigna x a y (y = 10) y luego incrementa x (x = 11)

- ++x es un preincremento: primero incrementa x (x = 12) y luego asigna ese valor a z (z = 12)

18. Identifique y corrija el error en el siguiente código recursivo que debería calcular el factorial de un número:

```csharp
public int Factorial(int n) {
    if (n == 0) {
        return 0;
    }
    return n * Factorial(n - 1);
}
```

- **Respuesta**: El error está en el caso base. Para factorial(0) debería devolver 1, no 0.
- *Corrección*:

```csharp
public int Factorial(int n) {
    if (n == 0) {
        return 1; // Caso base correcto: 0! = 1
    }
    return n * Factorial(n - 1);
}
```

SECCIÓN 3: CONOCIMIENTOS DE .NET Y C# (25 puntos)

3.1 Fundamentos de C#

19. ¿Cuál es la diferencia entre una clase abstracta y una interfaz en C#?
    - **Respuesta**:
      - **Clase abstracta**:
        - Puede contener implementaciones de métodos y variables de instancia
        - Puede tener constructores y destructores
        - Una clase solo puede heredar de una única clase abstracta
        - Los miembros pueden tener modificadores de acceso (public, private, protected)
        - Se utiliza cuando existe una relación "es un" entre clases
      - **Interfaz**:
        - Tradicionalmente solo define métodos sin implementación (aunque en C# moderno, desde 8.0, puede tener implementaciones predeterminadas)
        - No puede contener constructores, destructores o variables de instancia

- Una clase puede implementar múltiples interfaces
- Todos los miembros son implícitamente públicos
- Se utiliza cuando se quiere definir un contrato que varias clases pueden implementar

20. Explique la diferencia entre los tipos de datos `value type` y `reference type` en C#. Proporcione ejemplos de cada uno.
    - **Respuesta**:
        - **Value types**:
            - Se almacenan en la pila (stack)
            - Contienen directamente sus datos
            - Cuando se asignan a otra variable o se pasan como parámetros, se copia todo su valor
            - Ejemplos: int, float, double, bool, char, struct, enum
        - **Reference types**:
            - Se almacenan en el montículo (heap)
            - Contienen una referencia (puntero) a los datos
            - Cuando se asignan a otra variable o se pasan como parámetros, solo se copia la referencia
            - Ejemplos: class, interface, delegate, object, string, array

21. ¿Qué son los métodos de extensión en C#? Proporcione un ejemplo sencillo.
    - **Respuesta**: Los métodos de extensión permiten añadir métodos a tipos existentes sin modificar el tipo original ni crear un tipo derivado. Se implementan como métodos estáticos en clases estáticas, y utilizan la palabra clave `this` como primer parámetro.

    Ejemplo:

```csharp
public static class StringExtensions
{
    public static bool IsPalindrome(this string str)
    {
        string cleaned = new string(str.Where(c => Char.IsLetterOrDigit(c)).ToArray()).ToLo
        return cleaned.SequenceEqual(cleaned.Reverse());
    }
}

// Uso:
string texto = "Anita lava la tina";
bool esPalindromo = texto.IsPalindrome(); // true
```

3.2 Programación Orientada a Objetos

22. Explique los cuatro pilares de la Programación Orientada a Objetos y cómo se aplican en C#.
    - **Respuesta**:
        - **Encapsulamiento**: Ocultar los detalles internos de implementación y exponer solo lo necesario mediante:
            - Modificadores de acceso (public, private, protected, internal)
            - Propiedades con getters y setters
            - Ejemplo: `private int _count; public int Count { get => _count; set => _count = value; }`
        - **Herencia**: Permite que una clase herede atributos y métodos de otra clase:
            - En C# se implementa con `:` (ej. `class Hijo : Padre`)
            - C# solo permite herencia simple de clases
            - Soporta cadenas de herencia (A hereda de B, que hereda de C)
        - **Polimorfismo**: Permite que un objeto tome múltiples formas:
            - **Sobrecarga**: Múltiples métodos con el mismo nombre pero diferentes parámetros
            - **Sobreescritura**: Redefinir un método de la clase base usando `override`
            - **Interfaces**: Un objeto puede implementar múltiples interfaces
        - **Abstracción**: Representar características esenciales sin incluir los detalles:
            - Clases abstractas: `abstract class Animal { abstract void MakeSound(); }`
            - Interfaces: `interface IDrawable { void Draw(); }`

23. ¿Qué es la sobrecarga y la sobreescritura de métodos? Proporcione un ejemplo de cada uno.
    - **Respuesta**:
        - **Sobrecarga (Overloading)**: Definir múltiples métodos con el mismo nombre pero diferentes parámetros (tipo, número u orden).

```csharp
public class Calculator
{
    public int Add(int a, int b)
    {
        return a + b;
    }

    public double Add(double a, double b)
    {
        return a + b;
    }

    public int Add(int a, int b, int c)
    {
        return a + b + c;
    }
}
```

- **Sobreescritura (Overriding)**: Redefinir un método de la clase base en una clase derivada usando la palabra clave `override`.

```csharp
public class Shape
{
    public virtual double CalculateArea()
    {
        return 0;
    }
}

public class Circle : Shape
{
    public double Radius { get; set; }

    public override double CalculateArea()
    {
        return Math.PI * Radius * Radius;
    }
}
```

24. Explique qué es el polimorfismo y cómo se implementa en C#.
    - **Respuesta**: El polimorfismo es la capacidad de un objeto para comportarse de diferentes formas según el contexto. En C# se puede implementar de varias maneras:

1. **Polimorfismo en tiempo de compilación (estático)**:
   - Sobrecarga de métodos: Mismo nombre, diferentes parámetros
   - Sobrecarga de operadores: Redefinir el comportamiento de operadores (+, -, *, etc.)
2. **Polimorfismo en tiempo de ejecución (dinámico)**:
   - Sobreescritura de métodos: Usar virtual/override para redefinir métodos de la clase base
   - Interfaces: Múltiples clases pueden implementar la misma interfaz con comportamientos diferentes

Ejemplo de polimorfismo con interfaces:

```csharp
interface IAnimal
{
    string MakeSound();
}

class Dog : IAnimal
{
    public string MakeSound() => "Woof!";
}

class Cat : IAnimal
{
    public string MakeSound() => "Meow!";
}

// Uso polimórfico:
void PetSound(IAnimal animal)
{
    Console.WriteLine(animal.MakeSound());
}

// Llamadas:
PetSound(new Dog()); // Imprime "Woof!"
PetSound(new Cat()); // Imprime "Meow!"
```

3.3 ASP.NET y .NET Framework

25. ¿Cuál es la diferencia entre .NET Framework y .NET Core (ahora .NET)?

- **Respuesta**:
  - **.NET Framework**:
    - Solo para Windows
    - Monolítico (se instala completamente)

    - Incluye APIs específicas de Windows (como WinForms, WPF)
    - Código base cerrado
    - Actualizaciones vinculadas al sistema operativo
  - **.NET Core/.NET**:
    - Multiplataforma (Windows, Linux, macOS)
    - Modular (paquetes NuGet)
    - Optimizado para la nube y microservicios
    - Open-source
    - Mayor rendimiento
    - Implementación side-by-side (varias versiones pueden coexistir)

26. Explique brevemente el ciclo de vida de una solicitud HTTP en una aplicación ASP.NET MVC.
    - **Respuesta**: El ciclo de vida básico es:
      1. **Recepción de la solicitud**: El servidor web recibe la petición HTTP
      2. **Routing**: El sistema de enrutamiento determina qué controlador y acción manejarán la solicitud
      3. **Instanciación del controlador**: Se crea una instancia del controlador seleccionado
      4. **Model binding**: Los parámetros de la solicitud se vinculan a los parámetros del método de acción
      5. **Ejecución de la acción**: Se ejecuta el método de acción correspondiente
      6. **Ejecución del resultado**: La acción devuelve un resultado (View, JsonResult, etc.)
      7. **Renderización de la vista** (si aplica): Si el resultado es una vista, se procesa la plantilla Razor
      8. **Respuesta**: Se envía la respuesta HTTP al cliente

27. ¿Qué es la inyección de dependencias y cómo se implementa en .NET?
    - **Respuesta**: La inyección de dependencias (DI) es un patrón de diseño que permite proporcionar a una clase sus dependencias desde fuentes externas en lugar de que la clase las cree internamente. Esto promueve el acoplamiento débil, facilita las pruebas unitarias y mejora la modularidad.

    En .NET, se implementa nativamente mediante:
    1. **Registro de servicios**: En el método `ConfigureServices` del `Startup.cs`

```csharp
public void ConfigureServices(IServiceCollection services)
{
    services.AddTransient<IEmailService, SmtpEmailService>();
    services.AddScoped<IUserRepository, UserRepository>();
    services.AddSingleton<IConfiguration, AppConfiguration>();
}
```

2. **Consumo de servicios**: A través de constructores o inyección de métodos

```csharp
public class UserController : Controller
{
    private readonly IUserRepository _userRepository;
    private readonly IEmailService _emailService;

    public UserController(IUserRepository userRepository, IEmailService emailService)
    {
        _userRepository = userRepository;
        _emailService = emailService;
    }

    // Métodos que usan las dependencias
}
```

SECCIÓN 4: EJERCICIO PRÁCTICO (20 puntos)

28. Implemente una función en C# que reciba una cadena y devuelva la misma cadena pero con las palabras invertidas.

csharp

```csharp
public string ReverseWords(string input)
{
    if (string.IsNullOrEmpty(input))
    {
        return input;
    }

    char[] result = new char[input.Length];

    int wordStart = 0;
    for (int i = 0; i <= input.Length; i++)
    {
        // Detectar final de palabra (espacio o fin de cadena)
        if (i == input.Length || input[i] == ' ')
        {
            // Copiar la palabra invertida
            int wordEnd = i - 1;
            for (int j = wordEnd; j >= wordStart; j--)
            {
                result[wordStart + (wordEnd - j)] = input[j];
            }

            // Si no es el final de la cadena, copiar el espacio
            if (i < input.Length)
            {
                result[i] = ' ';
            }

            // Actualizar el inicio de la siguiente palabra
            wordStart = i + 1;
        }
    }

    return new string(result);
}
```

29. Implemente una clase `SimpleCache<T>` que almacene objetos con una clave asociada (string) y un tiempo de expiración.

csharp

```
public class SimpleCache<T>
{
    private class CacheItem
    {
        public T Value { get; set; }
        public DateTime ExpirationTime { get; set; }

        public bool IsExpired()
        {
            return DateTime.Now > ExpirationTime;
        }
    }

    private Dictionary<string, CacheItem> _cache = new Dictionary<string, CacheItem>();

    public void Add(string key, T value, int expirationTimeInSeconds)
    {
        if (string.IsNullOrEmpty(key))
        {
            throw new ArgumentException("La clave no puede ser nula o vacía", nameof(key));
        }

        var expirationTime = DateTime.Now.AddSeconds(expirationTimeInSeconds);
        var cacheItem = new CacheItem
        {
            Value = value,
            ExpirationTime = expirationTime
        };

        // Si la clave ya existe, la actualizamos
        if (_cache.ContainsKey(key))
        {
            _cache[key] = cacheItem;
        }
        else
        {
            _cache.Add(key, cacheItem);
        }
    }

    public T Get(string key)
    {
        if (string.IsNullOrEmpty(key) || !_cache.ContainsKey(key))
        {
            return default(T);
        }
```

```
        var cacheItem = _cache[key];

        if (cacheItem.IsExpired())
        {
            _cache.Remove(key);
            return default(T);
        }

        return cacheItem.Value;
    }

    public bool Remove(string key)
    {
        if (string.IsNullOrEmpty(key) || !_cache.ContainsKey(key))
        {
            return false;
        }

        return _cache.Remove(key);
    }

    public void CleanExpired()
    {
        var expiredKeys = _cache
            .Where(pair => pair.Value.IsExpired())
            .Select(pair => pair.Key)
            .ToList();

        foreach (var key in expiredKeys)
        {
            _cache.Remove(key);
        }
    }
}
```