Ejercicios de Recursión

Menor elemento

Dado un array que no está necesariamente ordenado, implemente el método `Min` que busca recursivamente el menor de los elementos.

```
int Min(int[] elements) {
    // Devuelve le menor de los elementos en el array
}
```

Búsqueda binaria

Dado un *array* ordenado de enteros, implemente el método `BinarySearch` que realiza una búsqueda binaria recursiva en el array.

```
bool BinarySearch(int[] array, int x) {
    // Devuelve true si el elemento `x` está en el array.
}
```

Factorial

El factorial de un número entero, $n!$, se define como la multiplicación de todos los números entre 1 y n:

$$n! = \Pi_{k=1}^{n} k$$

Una posible definición recursiva de $n!$ es la siguiente:

$$n! = n \cdot (n-1)!$$

Tenga en cuenta que por definición, $0! = 1$.

Implemente el método `Factorial` que computa el factorial de un número de forma recursiva.

```
int Factorial(int n) {
    // Devuelve n!
}
```

Fibonacci

La sucesión de Fibonacci se define recursivamente de la siguiente forma:

$$\begin{aligned} F(0) &= 1 \\ F(1) &= 1 \\ F(n) &= F(n-1) + F(n-2) \end{aligned}$$

Implemente el método `Fibonnaci` que calcula el n-ésimo elemento de la sucesión de Fibonnaci.

```
int Fibonacci(int n) {
    // Calcula el n-ésimo elemento de Fibonacci
}
```

Palindromo

Una cadena de caracteres es palíndromo si se lee igual de izquierda a derecha que de derecha a izquierda. Note que una cadena puede ser palíndromo tanto si es de longitud par como si es impar.

Implemente el método `EsPalindromo` que devuelve `true` si la cadena es palíndromo, de forma recursiva.

```
bool EsPalindromo(string s) {
    // Devuelve true is s es palíndromo
}
```

Recorrido del caballo

Se desea saber si existe alguna manera de recorrer un tablero de ajedrez de $n \times n$ celdas con un caballo, tal que empezando en la celda superior izquierda (digamos que es la celda $(0, 0)$) el caballo pase exactamente una vez por cada celda.

Recuerde que el movimiento del caballo es dos celdas en una dirección (horizontal o vertical) y luego una celda en una dirección ortogonal.

Implemente el método `HayRecorridoCaballo` que devuelve `true` si existe al menos una forma de hacer este recorrido en un tablero de tamaño $n \times n$:

```
bool HayRecorridoCaballo(int n) {
    // Devuelve true si existe al menos un recorrido
    // del caballo en un tablero de n x n
}
```

Implemente el método `RecorridosCaballo` que devuelve la cantidad de recorridos posibles en dicho tablero:

```
int RecorridosCaballo(int n) {
    // Devuelve la cantidad total de posibles recorridos
    // del caballo en un tablero de n x n
}
```

Ejercicios sobre POO

Filtrado genérico

Se tiene la interfaz genérica `IFilter` que encapsula el concepto de un criterio de filtrado. Aquellos elementos para los que el método `Apply` devuelve `true` se consideran que cumplen el criterio asociado.

```
interface IFilter<T> {
    bool Apply(T item);
}
```

A partir de esta interfaz, se quiere implementar la interfaz `IFiltering` que encapsula operaciones a realizar con filtros.

```
interface IFiltering<T> {
    int Count(T[] items, IFilter<T> filter) {
        // Devolver la cantidad de elementos en `items`
        // que cumplen con el criterio de filtro.
    }

    T[] Select(T[] items, IFilter<T> filter) {
        // Devolver los elementos en `items` para los que
        // se cumple el criterio.
    }

    IFilter<T> Complement(IFilter<T> filter) {
        // Devolver una implementación de `IFilter`
        // que encapsule el criterio complementario
        // de `filter`.
    }
}
```

Por ejemplo, una posible implementación es la clase `OddFilter` que representa el criterio de un número que es impar.

```
class OddFilter: IFilter<int> {
    public bool Apply(int item) {
        return item % 2 != 0;
    }
}
```

A partir de este filtro y una implementación sensible de `IFiltering` se podría hacer lo siguiente:

```
IFiltering<int> filtering = new MyFiltering<int>(); // su implementación
IFilter<int> oddFilter = new OddFilter();

int[] items = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

```
filtering.Count(items, oddFilter); // Devolvería 5

filtering.Select(items, oddFilter);
// Devolvería {1, 3, 5, 7, 9}

filtering.Select(items, filtering.Complement(oddFilter));
// Devolvería {0, 2, 4, 6, 8}
```