Lab 7 – programación con Yul

1. Programación con Yul y el *assembly block*

1. Estudia la función `completeMaxAsm` del fichero `09fors.sol` y a partir de ella programa una función Solidity:

```
function maxMinMemory(uint[] memory arr) public pure
returns (uint maxmin) {
  assembly {
    // ...
  }
}
```

que calcule la distancia entre el máximo y el mínimo del array `arr` **utilizando exclusivamente código Yul** dentro de `maxMinMemory`. Para ello, **debes definir una función Yul en un bloque `assembly`:**

```
function fmaxmin (array_pointer) -> maxVal, minVal
```

que devuelva el máximo y el mínimo del array proporcionado como argumento para después invocarla y calcular la diferencia entre los dos resultados en el mismo bloque Yul.

2. Dado el siguiente contrato:

```
contract lab6ex6 {
  uint[] public arr;

  function generate(uint n) external {
    // Populates the array with some weird small numbers.
    bytes32 b = keccak256("seed");
    for (uint i = 0; i < n; i++) {
      uint8 number = uint8(b[i % 32]);
      arr.push(number);
    }
  }


  function maxMinStorage() public view returns (uint maxmin){
    //...
  }
}
```

Escribe el cuerpo de la función `maxMinStorage` que realice el mismo cálculo que el apartado anterior para un array `arr` en *storage*, utilizando exclusivamente código Yul. Para ello, define una función Yul

```
function fmaxmin (slot) -> maxVal, minVal
```

que, dado un slot de storage que corresponde a un array, calcule el máximo y mínimo del array. Para obtener en Yul el slot de una variable de *storage* `arr`, se debe utilizar `arr.slot`. Intenta reducir al máximo el consumo de gas.

3. Crea otro contrato con una implementación simple de `maxMinStorage()` escrita en Solidity y compara el coste de ejecución respecto de la versión en Yul utilizando un array de 100 elementos. Analiza los resultados obtenidos y explica por qué tu implementación en Yul reduce el consumo de gas.