VRF Best Practices

Best are the practices for using Chainlink VRF.

Getting a random number within a range

If you need to generate a random number within a given range, you should use modulo to define the limits of your range. Below you can see how to get a random number between 1 and 50.

```
uint256 public randomResult;

function fulfillRandomness(bytes32 requestId, uint256 randomness) internal ove
    randomResult = (randomness % 50) + 1;
}
```

Getting multiple random numbers

If you want to get multiple random numbers from a single VRF response, you should create an array where the `randomValue` is your original returned VRF number and `n` is the desired number of random numbers.

```
function expand(uint256 randomValue, uint256 n) public pure returns (uint256[]
    expandedValues = new uint256[](n);
    for (uint256 i = 0; i < n; i++) {
        expandedValues[i] = uint256(keccak256(abi.encode(randomValue, i)));
    }
    return expandedValues;
}
```

Having multiple VRF requests in flight

If you want to have multiple VRF requests in flight, you might want to create a mapping between the `requestId` and the address of the requester.

```
mapping(bytes32 => address) public requestIdToAddress;

function getRandomNumber() public returns (bytes32 requestId) {
    require(LINK.balanceOf(address(this)) >= fee, "Not enough LINK - fill cont
    bytes32 requestId =  requestRandomness(keyHash, fee);
    requestIdToAddress[requestId] = msg.sender;
}

function fulfillRandomness(bytes32 requestId, uint256 randomness) internal ove
```

```
    address requestAddress = requestIdToAddress[requestId];
}
```

If you want to keep order when a request was made, you might want to use a mapping of `requestId` to the index/order of this request.

```
mapping(bytes32 => uint256) public requestIdToRequestNumberIndex;
uint256 public requestCounter;

function getRandomNumber() public returns (bytes32 requestId) {
    require(LINK.balanceOf(address(this)) >= fee, "Not enough LINK - fill contr
    bytes32 requestId =  requestRandomness(keyHash, fee);
    requestIdToRequestNumberIndex[requestId] = requestCounter;
    requestCounter += 1;
}

function fulfillRandomness(bytes32 requestId, uint256 randomness) internal over
    uint256 requestNumber = requestIdToRequestNumberIndex[requestId];
}
```

If you want to keep generated random numbers of several VRF requests, you might want to use a mapping of `requestId` to the returned random number.

Solidity (Ethereum) Copy

```
mapping(bytes32 => uint256) public requestIdToRandomNumber;

function getRandomNumber() public returns (bytes32 requestId) {
    require(LINK.balanceOf(address(this)) >= fee, "Not enough LINK - fill contr
    return requestRandomness(keyHash, fee);
}

function fulfillRandomness(bytes32 requestId, uint256 randomness) internal over
    requestIdToRandomNumber[requestId] = randomness;
}
```

Feel free to use whatever data structure you prefer.