- 0 Removals + 1 Addition

```
// SPDX-License-Identifier: AGPL-3.0-or-later
pragma solidity 0.7.5;

library LowGasSafeMath {
    /// @notice Returns x + y, reverts if sum overflows uint256
    /// @param x The augend
    /// @param y The addend
    /// @return z The sum of x and y
    function add(uint256 x, uint256 y) internal pure returns (uint256 z) {
        require((z = x + y) >= x);
    }

    function add32(uint32 x, uint32 y) internal pure returns (uint32 z) {
        require((z = x + y) >= x);
    }

    /// @notice Returns x - y, reverts if underflows
    /// @param x The minuend
    /// @param y The subtrahend
    /// @return z The difference of x and y
    function sub(uint256 x, uint256 y) internal pure returns (uint256 z) {
        require((z = x - y) <= x);
    }

    function sub32(uint32 x, uint32 y) internal pure returns (uint32 z) {
        require((z = x - y) <= x);
    }

    /// @notice Returns x * y, reverts if overflows
    /// @param x The multiplicand
    /// @param y The multiplier
    /// @return z The product of x and y
    function mul(uint256 x, uint256 y) internal pure returns (uint256 z) {
        require(x == 0 || (z = x * y) / x == y);
    }

    /// @notice Returns x + y, reverts if overflows or underflows
    /// @param x The augend
    /// @param y The addend
    /// @return z The sum of x and y
    function add(int256 x, int256 y) internal pure returns (int256 z) {
        require((z = x + y) >= x == (y >= 0));
    }

    /// @notice Returns x - y, reverts if overflows or underflows
    /// @param x The minuend
    /// @param y The subtrahend
    /// @return z The difference of x and y
    function sub(int256 x, int256 y) internal pure returns (int256 z) {
        require((z = x - y) <= x == (y >= 0));
    }
}
```

```
// SPDX-License-Identifier: AGPL-3.0-or-later
pragma solidity 0.7.5;
// Only change to generate diff
library LowGasSafeMath {
    /// @notice Returns x + y, reverts if sum overflows uint256
    /// @param x The augend
    /// @param y The addend
    /// @return z The sum of x and y
    function add(uint256 x, uint256 y) internal pure returns (uint256 z) {
        require((z = x + y) >= x);
    }

    function add32(uint32 x, uint32 y) internal pure returns (uint32 z) {
        require((z = x + y) >= x);
    }

    /// @notice Returns x - y, reverts if underflows
    /// @param x The minuend
    /// @param y The subtrahend
    /// @return z The difference of x and y
    function sub(uint256 x, uint256 y) internal pure returns (uint256 z) {
        require((z = x - y) <= x);
    }

    function sub32(uint32 x, uint32 y) internal pure returns (uint32 z) {
        require((z = x - y) <= x);
    }

    /// @notice Returns x * y, reverts if overflows
    /// @param x The multiplicand
    /// @param y The multiplier
    /// @return z The product of x and y
    function mul(uint256 x, uint256 y) internal pure returns (uint256 z) {
        require(x == 0 || (z = x * y) / x == y);
    }

    /// @notice Returns x + y, reverts if overflows or underflows
    /// @param x The augend
    /// @param y The addend
    /// @return z The sum of x and y
    function add(int256 x, int256 y) internal pure returns (int256 z) {
        require((z = x + y) >= x == (y >= 0));
    }

    /// @notice Returns x - y, reverts if overflows or underflows
    /// @param x The minuend
    /// @param y The subtrahend
    /// @return z The difference of x and y
    function sub(int256 x, int256 y) internal pure returns (int256 z) {
        require((z = x - y) <= x == (y >= 0));
    }
}
```

```

interface IERC20 {
    function decimals() external view returns (uint8);
  /**
   * @dev Returns the amount of tokens in existence.
   */
  function totalSupply() external view returns (uint256);

  /**
   * @dev Returns the amount of tokens owned by `account`.
   */
  function balanceOf(address account) external view returns (uint256);

  /**
   * @dev Moves `amount` tokens from the caller's account to `recipient`.
   *
   * Returns a boolean value indicating whether the operation succeeded.
   *
   * Emits a {Transfer} event.
   */
  function transfer(address recipient, uint256 amount) external returns (bool);

  /**
   * @dev Returns the remaining number of tokens that `spender` will be
   * allowed to spend on behalf of `owner` through {transferFrom}. This is
   * zero by default.
   *
   * This value changes when {approve} or {transferFrom} are called.
   */
  function allowance(address owner, address spender) external view returns (uint256);

  /**
   * @dev Sets `amount` as the allowance of `spender` over the caller's tokens.
   *
   * Returns a boolean value indicating whether the operation succeeded.
   *
   * IMPORTANT: Beware that changing an allowance with this method brings the risk
   * that someone may use both the old and the new allowance by unfortunate
   * transaction ordering. One possible solution to mitigate this race
   * condition is to first reduce the spender's allowance to 0 and set the
   * desired value afterwards:
   * https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
   *
   * Emits an {Approval} event.
   */
  function approve(address spender, uint256 amount) external returns (bool);

  /**
```

```

interface IERC20 {
    function decimals() external view returns (uint8);
  /**
   * @dev Returns the amount of tokens in existence.
   */
  function totalSupply() external view returns (uint256);

  /**
   * @dev Returns the amount of tokens owned by `account`.
   */
  function balanceOf(address account) external view returns (uint256);

  /**
   * @dev Moves `amount` tokens from the caller's account to `recipient`.
   *
   * Returns a boolean value indicating whether the operation succeeded.
   *
   * Emits a {Transfer} event.
   */
  function transfer(address recipient, uint256 amount) external returns (bool);

  /**
   * @dev Returns the remaining number of tokens that `spender` will be
   * allowed to spend on behalf of `owner` through {transferFrom}. This is
   * zero by default.
   *
   * This value changes when {approve} or {transferFrom} are called.
   */
  function allowance(address owner, address spender) external view returns (uint256);

  /**
   * @dev Sets `amount` as the allowance of `spender` over the caller's tokens.
   *
   * Returns a boolean value indicating whether the operation succeeded.
   *
   * IMPORTANT: Beware that changing an allowance with this method brings the risk
   * that someone may use both the old and the new allowance by unfortunate
   * transaction ordering. One possible solution to mitigate this race
   * condition is to first reduce the spender's allowance to 0 and set the
   * desired value afterwards:
   * https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
   *
   * Emits an {Approval} event.
   */
  function approve(address spender, uint256 amount) external returns (bool);

  /**
```

```
   * @dev Moves `amount` tokens from `sender` to `recipient` using the
   * allowance mechanism. `amount` is then deducted from the caller's
   * allowance.
   *
   * Returns a boolean value indicating whether the operation succeeded.
   *
   * Emits a {Transfer} event.
   */
  function transferFrom(address sender, address recipient, uint256 amount) external returns (bool);

  /**
   * @dev Emitted when `value` tokens are moved from one account (`from`) to
   * another (`to`).
   *
   * Note that `value` may be zero.
   */
  event Transfer(address indexed from, address indexed to, uint256 value);

  /**
   * @dev Emitted when the allowance of a `spender` for an `owner` is set by
   * a call to {approve}. `value` is the new allowance.
   */
  event Approval(address indexed owner, address indexed spender, uint256 value);
}

library Address {
    /**
     * @dev Returns true if `account` is a contract.
     *
     * [IMPORTANT]
     * ====
     * It is unsafe to assume that an address for which this function returns
     * false is an externally-owned account (EOA) and not a contract.
     *
     * Among others, `isContract` will return false for the following
     * types of addresses:
     *
     *  - an externally-owned account
     *  - a contract in construction
     *  - an address where a contract will be created
     *  - an address where a contract lived, but was destroyed
     * ====
     */
    function isContract(address account) internal view returns (bool) {
        // This method relies in extcodesize, which returns 0 for contracts in
        // construction, since the code is only stored at the end of the
        // constructor execution.

        uint256 size;
```

```
   * @dev Moves `amount` tokens from `sender` to `recipient` using the
   * allowance mechanism. `amount` is then deducted from the caller's
   * allowance.
   *
   * Returns a boolean value indicating whether the operation succeeded.
   *
   * Emits a {Transfer} event.
   */
  function transferFrom(address sender, address recipient, uint256 amount) external returns (bool);

  /**
   * @dev Emitted when `value` tokens are moved from one account (`from`) to
   * another (`to`).
   *
   * Note that `value` may be zero.
   */
  event Transfer(address indexed from, address indexed to, uint256 value);

  /**
   * @dev Emitted when the allowance of a `spender` for an `owner` is set by
   * a call to {approve}. `value` is the new allowance.
   */
  event Approval(address indexed owner, address indexed spender, uint256 value);
}

library Address {
    /**
     * @dev Returns true if `account` is a contract.
     *
     * [IMPORTANT]
     * ====
     * It is unsafe to assume that an address for which this function returns
     * false is an externally-owned account (EOA) and not a contract.
     *
     * Among others, `isContract` will return false for the following
     * types of addresses:
     *
     *  - an externally-owned account
     *  - a contract in construction
     *  - an address where a contract will be created
     *  - an address where a contract lived, but was destroyed
     * ====
     */
    function isContract(address account) internal view returns (bool) {
        // This method relies in extcodesize, which returns 0 for contracts in
        // construction, since the code is only stored at the end of the
        // constructor execution.

        uint256 size;
        // solhint-disable-next-line no-inline-assembly
        assembly { size := extcodesize(account) }
        return size > 0;
    }

    /**
     * @dev Replacement for Solidity's `transfer`: sends `amount` wei to
     * `recipient`, forwarding all available gas and reverting on errors.
     *
     * https://eips.ethereum.org/EIPS/eip-1884[EIP1884] increases the gas cost
     * of certain opcodes, possibly making contracts go over the 2300 gas limit
     * imposed by `transfer`, making them unable to receive funds via
     * `transfer`. {sendValue} removes this limitation.
     *
     * https://diligence.consensys.net/posts/2019/09/stop-using-soliditys-transfer-now/[Learn more].
     *
     * IMPORTANT: because control is transferred to `recipient`, care must be
     * taken to not create reentrancy vulnerabilities. Consider using
     * {ReentrancyGuard}
     */
    function sendValue(address payable recipient, uint256 amount) internal {
        require(address(this).balance >= amount, "Address: insufficient balance");

        // solhint-disable-next-line avoid-low-level-calls, avoid-call-value
        (bool success, ) = recipient.call{ value: amount }("");
        require(success, "Address: unable to send value, recipient may have reverted");
    }

    /**
     * @dev Performs a Solidity function call using a low level `call`. A
     * plain`call` is an unsafe replacement for a function call: use this
     * function instead.
     *
     * If `target` reverts with a revert reason, it is bubbled up by this
     * function (like regular Solidity function calls).
     *
     *
     * Requirements:
     *
     * - `target` must be a contract.
     * - calling `target` with `data` must not revert.
     *
     * _Available since v3.1._
     */
    function functionCall(address target, bytes memory data) internal returns (bytes memory) {
      return functionCall(target, data, "Address: low-level call failed");
```

```
        // solhint-disable-next-line no-inline-assembly
        assembly { size := extcodesize(account) }
        return size > 0;
    }

    /**
     * @dev Replacement for Solidity's `transfer`: sends `amount` wei to
     * `recipient`, forwarding all available gas and reverting on errors.
     *
     * https://eips.ethereum.org/EIPS/eip-1884[EIP1884] increases the gas cost
     * of certain opcodes, possibly making contracts go over the 2300 gas limit
     * imposed by `transfer`, making them unable to receive funds via
     * `transfer`. {sendValue} removes this limitation.
     *
     * https://diligence.consensys.net/posts/2019/09/stop-using-soliditys-transfer-now/[Learn more].
     *
     * IMPORTANT: because control is transferred to `recipient`, care must be
     * taken to not create reentrancy vulnerabilities. Consider using
     * {ReentrancyGuard}
     */
    function sendValue(address payable recipient, uint256 amount) internal {
        require(address(this).balance >= amount, "Address: insufficient balance");

        // solhint-disable-next-line avoid-low-level-calls, avoid-call-value
        (bool success, ) = recipient.call{ value: amount }("");
        require(success, "Address: unable to send value, recipient may have reverted");
    }

    /**
     * @dev Performs a Solidity function call using a low level `call`. A
     * plain`call` is an unsafe replacement for a function call: use this
     * function instead.
     *
     * If `target` reverts with a revert reason, it is bubbled up by this
     * function (like regular Solidity function calls).
     *
     *
     * Requirements:
     *
     * - `target` must be a contract.
     * - calling `target` with `data` must not revert.
     *
     * _Available since v3.1._
     */
    function functionCall(address target, bytes memory data) internal returns (bytes memory) {
      return functionCall(target, data, "Address: low-level call failed");
    }

    /**
     * @dev Same as {xref-Address-functionCall-address-bytes-}[`functionCall`], but with
     * `errorMessage` as a fallback revert reason when `target` reverts.
     *
     * _Available since v3.1._
     */
    function functionCall(
        address target,
        bytes memory data,
        string memory errorMessage
    ) internal returns (bytes memory) {
        return _functionCallWithValue(target, data, 0, errorMessage);
    }

    /**
     * @dev Same as {xref-Address-functionCall-address-bytes-}[`functionCall`],
     * but also transferring `value` wei to `target`.
     *
     * Requirements:
     *
     * - the calling contract must have an ETH balance of at least `value`.
     * - the called Solidity function must be `payable`.
     *
     * _Available since v3.1._
     */
    function functionCallWithValue(address target, bytes memory data, uint256 value) internal returns (bytes memory) {
        return functionCallWithValue(target, data, value, "Address: low-level call with value failed");
    }

    /**
     * @dev Same as {xref-Address-functionCallWithValue-address-bytes-uint256-}[`functionCallWithValue`], but
     * with `errorMessage` as a fallback revert reason when `target` reverts.
     *
     * _Available since v3.1._
     */
    function functionCallWithValue(
        address target,
        bytes memory data,
        uint256 value,
        string memory errorMessage
    ) internal returns (bytes memory) {
        require(address(this).balance >= value, "Address: insufficient balance for call");
        require(isContract(target), "Address: call to non-contract");

        // solhint-disable-next-line avoid-low-level-calls
        (bool success, bytes memory returndata) = target.call{ value: value }(data);
        return _verifyCallResult(success, returndata, errorMessage);
```

```
    }

    /**
     * @dev Same as {xref-Address-functionCall-address-bytes-}[`functionCall`], but with
     * `errorMessage` as a fallback revert reason when `target` reverts.
     *
     * _Available since v3.1._
     */
    function functionCall(
        address target,
        bytes memory data,
        string memory errorMessage
    ) internal returns (bytes memory) {
        return _functionCallWithValue(target, data, 0, errorMessage);
    }

    /**
     * @dev Same as {xref-Address-functionCall-address-bytes-}[`functionCall`],
     * but also transferring `value` wei to `target`.
     *
     * Requirements:
     *
     * - the calling contract must have an ETH balance of at least `value`.
     * - the called Solidity function must be `payable`.
     *
     * _Available since v3.1._
     */
    function functionCallWithValue(address target, bytes memory data, uint256 value) internal returns (bytes memory) {
        return functionCallWithValue(target, data, value, "Address: low-level call with value failed");
    }

    /**
     * @dev Same as {xref-Address-functionCallWithValue-address-bytes-uint256-}[`functionCallWithValue`], but
     * with `errorMessage` as a fallback revert reason when `target` reverts.
     *
     * _Available since v3.1._
     */
    function functionCallWithValue(
        address target,
        bytes memory data,
        uint256 value,
        string memory errorMessage
    ) internal returns (bytes memory) {
        require(address(this).balance >= value, "Address: insufficient balance for call");
        require(isContract(target), "Address: call to non-contract");

        // solhint-disable-next-line avoid-low-level-calls
        (bool success, bytes memory returndata) = target.call{ value: value }(data);
        return _verifyCallResult(success, returndata, errorMessage);
    }

    function _functionCallWithValue(
        address target,
        bytes memory data,
        uint256 weiValue,
        string memory errorMessage
    ) private returns (bytes memory) {
        require(isContract(target), "Address: call to non-contract");

        // solhint-disable-next-line avoid-low-level-calls
        (bool success, bytes memory returndata) = target.call{ value: weiValue }(data);
        if (success) {
            return returndata;
        } else {
            // Look for revert reason and bubble it up if present
            if (returndata.length > 0) {
                // The easiest way to bubble the revert reason is using memory via assembly

                // solhint-disable-next-line no-inline-assembly
                assembly {
                    let returndata_size := mload(returndata)
                    revert(add(32, returndata), returndata_size)
                }
            } else {
                revert(errorMessage);
            }
        }
    }

  /**
     * @dev Same as {xref-Address-functionCall-address-bytes-}[`functionCall`],
     * but performing a static call.
     *
     * _Available since v3.3._
     */
    function functionStaticCall(address target, bytes memory data) internal view returns (bytes memory) {
        return functionStaticCall(target, data, "Address: low-level static call failed");
    }

    /**
     * @dev Same as {xref-Address-functionCall-address-bytes-string-}[`functionCall`],
     * but performing a static call.
     *
     * _Available since v3.3._
     */
    function functionStaticCall(
        address target,
        bytes memory data,
        string memory errorMessage
    ) internal view returns (bytes memory) {
        require(isContract(target), "Address: static call to non-contract");

        // solhint-disable-next-line avoid-low-level-calls
```

```
    }

    function _functionCallWithValue(
        address target,
        bytes memory data,
        uint256 weiValue,
        string memory errorMessage
    ) private returns (bytes memory) {
        require(isContract(target), "Address: call to non-contract");

        // solhint-disable-next-line avoid-low-level-calls
        (bool success, bytes memory returndata) = target.call{ value: weiValue }(data);
        if (success) {
            return returndata;
        } else {
            // Look for revert reason and bubble it up if present
            if (returndata.length > 0) {
                // The easiest way to bubble the revert reason is using memory via assembly

                // solhint-disable-next-line no-inline-assembly
                assembly {
                    let returndata_size := mload(returndata)
                    revert(add(32, returndata), returndata_size)
                }
            } else {
                revert(errorMessage);
            }
        }
    }

  /**
     * @dev Same as {xref-Address-functionCall-address-bytes-}[`functionCall`],
     * but performing a static call.
     *
     * _Available since v3.3._
     */
    function functionStaticCall(address target, bytes memory data) internal view returns (bytes memory) {
        return functionStaticCall(target, data, "Address: low-level static call failed");
    }

    /**
     * @dev Same as {xref-Address-functionCall-address-bytes-string-}[`functionCall`],
     * but performing a static call.
     *
     * _Available since v3.3._
     */
    function functionStaticCall(
        address target,
        bytes memory data,
        string memory errorMessage
    ) internal view returns (bytes memory) {
        require(isContract(target), "Address: static call to non-contract");

        // solhint-disable-next-line avoid-low-level-calls
        (bool success, bytes memory returndata) = target.staticcall(data);
        return _verifyCallResult(success, returndata, errorMessage);
    }

    /**
     * @dev Same as {xref-Address-functionCall-address-bytes-}[`functionCall`],
     * but performing a delegate call.
     *
     * _Available since v3.3._
     */
    function functionDelegateCall(address target, bytes memory data) internal returns (bytes memory) {
        return functionDelegateCall(target, data, "Address: low-level delegate call failed");
    }

    /**
     * @dev Same as {xref-Address-functionCall-address-bytes-string-}[`functionCall`],
     * but performing a delegate call.
     *
     * _Available since v3.3._
     */
    function functionDelegateCall(
        address target,
        bytes memory data,
        string memory errorMessage
    ) internal returns (bytes memory) {
        require(isContract(target), "Address: delegate call to non-contract");

        // solhint-disable-next-line avoid-low-level-calls
        (bool success, bytes memory returndata) = target.delegatecall(data);
        return _verifyCallResult(success, returndata, errorMessage);
    }

    function _verifyCallResult(
        bool success,
        bytes memory returndata,
        string memory errorMessage
    ) private pure returns(bytes memory) {
        if (success) {
            return returndata;
        } else {
            // Look for revert reason and bubble it up if present
            if (returndata.length > 0) {
                // The easiest way to bubble the revert reason is using memory via assembly

                // solhint-disable-next-line no-inline-assembly
                assembly {
                    let returndata_size := mload(returndata)
                    revert(add(32, returndata), returndata_size)
                }
            } else {
                revert(errorMessage);
            }
        }
```

```
        (bool success, bytes memory returndata) = target.staticcall(data);
        return _verifyCallResult(success, returndata, errorMessage);
    }

    /**
     * @dev Same as {xref-Address-functionCall-address-bytes-}[`functionCall`],
     * but performing a delegate call.
     *
     * _Available since v3.3._
     */
    function functionDelegateCall(address target, bytes memory data) internal returns (bytes memory) {
        return functionDelegateCall(target, data, "Address: low-level delegate call failed");
    }

    /**
     * @dev Same as {xref-Address-functionCall-address-bytes-string-}[`functionCall`],
     * but performing a delegate call.
     *
     * _Available since v3.3._
     */
    function functionDelegateCall(
        address target,
        bytes memory data,
        string memory errorMessage
    ) internal returns (bytes memory) {
        require(isContract(target), "Address: delegate call to non-contract");

        // solhint-disable-next-line avoid-low-level-calls
        (bool success, bytes memory returndata) = target.delegatecall(data);
        return _verifyCallResult(success, returndata, errorMessage);
    }

    function _verifyCallResult(
        bool success,
        bytes memory returndata,
        string memory errorMessage
    ) private pure returns(bytes memory) {
        if (success) {
            return returndata;
        } else {
            // Look for revert reason and bubble it up if present
            if (returndata.length > 0) {
                // The easiest way to bubble the revert reason is using memory via assembly

                // solhint-disable-next-line no-inline-assembly
                assembly {
                    let returndata_size := mload(returndata)
                    revert(add(32, returndata), returndata_size)
                }
            } else {
                revert(errorMessage);
            }
        }
    }

    function addressToString(address _address) internal pure returns(string memory) {
        bytes32 _bytes = bytes32(uint256(_address));
        bytes memory HEX = "0123456789abcdef";
        bytes memory _addr = new bytes(42);

        _addr[0] = '0';
        _addr[1] = 'x';

        for(uint256 i = 0; i < 20; i++) {
            _addr[2+i*2] = HEX[uint8(_bytes[i + 12] >> 4)];
            _addr[3+i*2] = HEX[uint8(_bytes[i + 12] & 0x0f)];
        }

        return string(_addr);

    }
}

library SafeERC20 {
    using LowGasSafeMath for uint256;
    using Address for address;

    function safeTransfer(IERC20 token, address to, uint256 value) internal {
        _callOptionalReturn(token, abi.encodeWithSelector(token.transfer.selector, to, value));
    }

    function safeTransferFrom(IERC20 token, address from, address to, uint256 value) internal {
        _callOptionalReturn(token, abi.encodeWithSelector(token.transferFrom.selector, from, to, value));
    }

    /**
     * @dev Deprecated. This function has issues similar to the ones found in
     * {IERC20-approve}, and its usage is discouraged.
     *
     * Whenever possible, use {safeIncreaseAllowance} and
     * {safeDecreaseAllowance} instead.
     */
    function safeApprove(IERC20 token, address spender, uint256 value) internal {
        // safeApprove should only be called when setting an initial allowance,
        // or when resetting it to zero. To increase and decrease it, use
        // 'safeIncreaseAllowance' and 'safeDecreaseAllowance'
        // solhint-disable-next-line max-line-length
        require((value == 0) || (token.allowance(address(this), spender) == 0),
            "SafeERC20: approve from non-zero to non-zero allowance"
        );
        _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector, spender, value));
    }

```

```
    }

    function addressToString(address _address) internal pure returns(string memory) {
        bytes32 _bytes = bytes32(uint256(_address));
        bytes memory HEX = "0123456789abcdef";
        bytes memory _addr = new bytes(42);

        _addr[0] = '0';
        _addr[1] = 'x';

        for(uint256 i = 0; i < 20; i++) {
            _addr[2+i*2] = HEX[uint8(_bytes[i + 12] >> 4)];
            _addr[3+i*2] = HEX[uint8(_bytes[i + 12] & 0x0f)];
        }

        return string(_addr);

    }
}

library SafeERC20 {
    using LowGasSafeMath for uint256;
    using Address for address;

    function safeTransfer(IERC20 token, address to, uint256 value) internal {
        _callOptionalReturn(token, abi.encodeWithSelector(token.transfer.selector, to, value));
    }

    function safeTransferFrom(IERC20 token, address from, address to, uint256 value) internal {
        _callOptionalReturn(token, abi.encodeWithSelector(token.transferFrom.selector, from, to, value));
    }

    /**
     * @dev Deprecated. This function has issues similar to the ones found in
     * {IERC20-approve}, and its usage is discouraged.
     *
     * Whenever possible, use {safeIncreaseAllowance} and
     * {safeDecreaseAllowance} instead.
     */
    function safeApprove(IERC20 token, address spender, uint256 value) internal {
        // safeApprove should only be called when setting an initial allowance,
        // or when resetting it to zero. To increase and decrease it, use
        // 'safeIncreaseAllowance' and 'safeDecreaseAllowance'
        // solhint-disable-next-line max-line-length
        require((value == 0) || (token.allowance(address(this), spender) == 0),
            "SafeERC20: approve from non-zero to non-zero allowance"
        );
        _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector, spender, value));
    }

    function safeIncreaseAllowance(IERC20 token, address spender, uint256 value) internal {
        uint256 newAllowance = token.allowance(address(this), spender).add(value);
        _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector, spender, newAllowance));
    }

    function safeDecreaseAllowance(
        IERC20 token,
        address spender,
        uint256 value
    ) internal {
        uint256 newAllowance = token.allowance(address(this), spender)
            .sub(value);
        _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector, spender, newAllowance));
    }

    /**
     * @dev Imitates a Solidity high-level call (i.e. a regular function call to a contract), relaxing the requirement
     * on the return value: the return value is optional (but if data is returned, it must not be false).
     * @param token The token targeted by the call.
     * @param data The call data (encoded using abi.encode or one of its variants).
     */
    function _callOptionalReturn(IERC20 token, bytes memory data) private {
        // We need to perform a low level call here, to bypass Solidity's return data size checking mechanism, since
        // we're implementing it ourselves. We use {Address.functionCall} to perform this call, which verifies that
        // the target address contains contract code and also asserts for success in the low-level call.

        bytes memory returndata = address(token).functionCall(data, "SafeERC20: low-level call failed");
        if (returndata.length > 0) { // Return data is optional
            // solhint-disable-next-line max-line-length
            require(abi.decode(returndata, (bool)), "SafeERC20: ERC20 operation did not succeed");
        }
    }
}

contract OwnableData {
    address public owner;
    address public pendingOwner;
}

contract Ownable is OwnableData {
    event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);

    /// @notice `owner` defaults to msg.sender on construction.
```

```
    function safeIncreaseAllowance(IERC20 token, address spender, uint256 value) internal {
        uint256 newAllowance = token.allowance(address(this), spender).add(value);
        _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector, spender, newAllowance));
    }

    function safeDecreaseAllowance(
        IERC20 token,
        address spender,
        uint256 value
    ) internal {
        uint256 newAllowance = token.allowance(address(this), spender)
            .sub(value);
        _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector, spender, newAllowance));
    }

    /**
     * @dev Imitates a Solidity high-level call (i.e. a regular function call to a contract), relaxing the requirement
     * on the return value: the return value is optional (but if data is returned, it must not be false).
     * @param token The token targeted by the call.
     * @param data The call data (encoded using abi.encode or one of its variants).
     */
    function _callOptionalReturn(IERC20 token, bytes memory data) private {
        // We need to perform a low level call here, to bypass Solidity's return data size checking mechanism, since
        // we're implementing it ourselves. We use {Address.functionCall} to perform this call, which verifies that
        // the target address contains contract code and also asserts for success in the low-level call.

        bytes memory returndata = address(token).functionCall(data, "SafeERC20: low-level call failed");
        if (returndata.length > 0) { // Return data is optional
            // solhint-disable-next-line max-line-length
            require(abi.decode(returndata, (bool)), "SafeERC20: ERC20 operation did not succeed");
        }
    }
}

contract OwnableData {
    address public owner;
    address public pendingOwner;
}

contract Ownable is OwnableData {
    event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);

    /// @notice `owner` defaults to msg.sender on construction.
    constructor() {
        owner = msg.sender;
        emit OwnershipTransferred(address(0), msg.sender);
    }

    /// @notice Transfers ownership to `newOwner`. Either directly or claimable by the new pending owner.
    /// Can only be invoked by the current `owner`.
    /// @param newOwner Address of the new owner.
    /// @param direct True if `newOwner` should be set immediately. False if `newOwner` needs to use `claimOwnership`.
    /// @param renounce Allows the `newOwner` to be `address(0)` if `direct` and `renounce` is True. Has no effect otherwise.
    function transferOwnership(
        address newOwner,
        bool direct,
        bool renounce
    ) public onlyOwner {
        if (direct) {
            // Checks
            require(newOwner != address(0) || renounce, "Ownable: zero address");

            // Effects
            emit OwnershipTransferred(owner, newOwner);
            owner = newOwner;
            pendingOwner = address(0);
        } else {
            // Effects
            pendingOwner = newOwner;
        }
    }

    /// @notice Needs to be called by `pendingOwner` to claim ownership.
    function claimOwnership() public {
        address _pendingOwner = pendingOwner;

        // Checks
        require(msg.sender == _pendingOwner, "Ownable: caller != pending owner");

        // Effects
        emit OwnershipTransferred(owner, _pendingOwner);
        owner = _pendingOwner;
        pendingOwner = address(0);
    }

    /// @notice Only allows the `owner` to execute the function.
    modifier onlyOwner() {
        require(msg.sender == owner, "Ownable: caller is not the owner");
        _;
    }
}

interface IMemo is IERC20 {
    function rebase( uint256 ohmProfit_, uint epoch_) external returns (uint256);

    function circulatingSupply() external view returns (uint256);
```

```
    constructor() {
        owner = msg.sender;
        emit OwnershipTransferred(address(0), msg.sender);
    }

    /// @notice Transfers ownership to `newOwner`. Either directly or claimable by the new pending owner.
    /// Can only be invoked by the current `owner`.
    /// @param newOwner Address of the new owner.
    /// @param direct True if `newOwner` should be set immediately. False if `newOwner` needs to use `claimOwnership`.
    /// @param renounce Allows the `newOwner` to be `address(0)` if `direct` and `renounce` is True. Has no effect otherwise.
    function transferOwnership(
        address newOwner,
        bool direct,
        bool renounce
    ) public onlyOwner {
        if (direct) {
            // Checks
            require(newOwner != address(0) || renounce, "Ownable: zero address");

            // Effects
            emit OwnershipTransferred(owner, newOwner);
            owner = newOwner;
            pendingOwner = address(0);
        } else {
            // Effects
            pendingOwner = newOwner;
        }
    }

    /// @notice Needs to be called by `pendingOwner` to claim ownership.
    function claimOwnership() public {
        address _pendingOwner = pendingOwner;

        // Checks
        require(msg.sender == _pendingOwner, "Ownable: caller != pending owner");

        // Effects
        emit OwnershipTransferred(owner, _pendingOwner);
        owner = _pendingOwner;
        pendingOwner = address(0);
    }

    /// @notice Only allows the `owner` to execute the function.
    modifier onlyOwner() {
        require(msg.sender == owner, "Ownable: caller is not the owner");
        _;
    }
}

interface IMemo is IERC20 {
    function rebase( uint256 ohmProfit_, uint epoch_) external returns (uint256);

    function circulatingSupply() external view returns (uint256);
```

```solidity

    function balanceOf(address who) external view override returns (uint256);

    function gonsForBalance( uint amount ) external view returns ( uint );

    function balanceForGons( uint gons ) external view returns ( uint );

    function index() external view returns ( uint );
}

interface IWarmup {
    function retrieve( address staker_, uint amount_ ) external;
}

interface IDistributor {
    function distribute() external returns ( bool );
}

contract TimeStaking is Ownable {

    using LowGasSafeMath for uint256;
    using LowGasSafeMath for uint32;
    using SafeERC20 for IERC20;
    using SafeERC20 for IMemo;

    IERC20 public immutable Time;
    IMemo public immutable Memories;

    struct Epoch {
        uint number;
        uint distribute;
        uint32 length;
        uint32 endTime;
    }
    Epoch public epoch;

    IDistributor public distributor;

    uint public totalBonus;

    IWarmup public warmupContract;
    uint public warmupPeriod;

    event LogStake(address indexed recipient, uint256 amount);
    event LogClaim(address indexed recipient, uint256 amount);
    event LogForfeit(address indexed recipient, uint256 memoAmount, uint256 timeAmount);
    event LogDepositLock(address indexed user, bool locked);
    event LogUnstake(address indexed recipient, uint256 amount);
    event LogRebase(uint256 distribute);
    event LogSetContract(CONTRACTS contractType, address indexed _contract);
    event LogWarmupPeriod(uint period);

    constructor (
        address _Time,
        address _Memories,
        uint32 _epochLength,
        uint _firstEpochNumber,
        uint32 _firstEpochTime
```

```solidity

    function balanceOf(address who) external view override returns (uint256);

    function gonsForBalance( uint amount ) external view returns ( uint );

    function balanceForGons( uint gons ) external view returns ( uint );

    function index() external view returns ( uint );
}

interface IWarmup {
    function retrieve( address staker_, uint amount_ ) external;
}

interface IDistributor {
    function distribute() external returns ( bool );
}

contract TimeStaking is Ownable {

    using LowGasSafeMath for uint256;
    using LowGasSafeMath for uint32;
    using SafeERC20 for IERC20;
    using SafeERC20 for IMemo;

    IERC20 public immutable Time;
    IMemo public immutable Memories;

    struct Epoch {
        uint number;
        uint distribute;
        uint32 length;
        uint32 endTime;
    }
    Epoch public epoch;

    IDistributor public distributor;

    uint public totalBonus;

    IWarmup public warmupContract;
    uint public warmupPeriod;

    event LogStake(address indexed recipient, uint256 amount);
    event LogClaim(address indexed recipient, uint256 amount);
    event LogForfeit(address indexed recipient, uint256 memoAmount, uint256 timeAmount);
    event LogDepositLock(address indexed user, bool locked);
    event LogUnstake(address indexed recipient, uint256 amount);
    event LogRebase(uint256 distribute);
    event LogSetContract(CONTRACTS contractType, address indexed _contract);
    event LogWarmupPeriod(uint period);

    constructor (
        address _Time,
        address _Memories,
        uint32 _epochLength,
        uint _firstEpochNumber,
        uint32 _firstEpochTime
    ) {
        require( _Time != address(0) );
        Time = IERC20(_Time);
        require( _Memories != address(0) );
        Memories = IMemo(_Memories);

        epoch = Epoch({
            length: _epochLength,
            number: _firstEpochNumber,
            endTime: _firstEpochTime,
            distribute: 0
        });
    }

    struct Claim {
        uint deposit;
        uint gons;
        uint expiry;
        bool lock; // prevents malicious delays
    }
    mapping( address => Claim ) public warmupInfo;

    /**
        @notice stake Time to enter warmup
        @param _amount uint
        @return bool
     */
    function stake( uint _amount, address _recipient ) external returns ( bool ) {
        rebase();

        Time.safeTransferFrom( msg.sender, address(this), _amount );

        Claim memory info = warmupInfo[ _recipient ];
        require( !info.lock, "Deposits for account are locked" );

        warmupInfo[ _recipient ] = Claim ({
            deposit: info.deposit.add( _amount ),
            gons: info.gons.add( Memories.gonsForBalance( _amount ) ),
            expiry: epoch.number.add( warmupPeriod ),
            lock: false
        });

        Memories.safeTransfer( address(warmupContract), _amount );
        emit LogStake(_recipient, _amount);
        return true;
    }

    /**
        @notice retrieve MEMO from warmup
        @param _recipient address
     */
    function claim ( address _recipient ) external {
        Claim memory info = warmupInfo[ _recipient ];
        if ( epoch.number >= info.expiry && info.expiry != 0 ) {
            delete warmupInfo[ _recipient ];
            uint256 amount = Memories.balanceForGons( info.gons );
```

```
    ) {
        require( _Time != address(0) );
        Time = IERC20(_Time);
        require( _Memories != address(0) );
        Memories = IMemo(_Memories);

        epoch = Epoch({
            length: _epochLength,
            number: _firstEpochNumber,
            endTime: _firstEpochTime,
            distribute: 0
        });
    }

    struct Claim {
        uint deposit;
        uint gons;
        uint expiry;
        bool lock; // prevents malicious delays
    }
    mapping( address => Claim ) public warmupInfo;

    /**
        @notice stake Time to enter warmup
        @param _amount uint
        @return bool
     */
    function stake( uint _amount, address _recipient ) external returns ( bool ) {
        rebase();

        Time.safeTransferFrom( msg.sender, address(this), _amount );

        Claim memory info = warmupInfo[ _recipient ];
        require( !info.lock, "Deposits for account are locked" );

        warmupInfo[ _recipient ] = Claim ({
            deposit: info.deposit.add( _amount ),
            gons: info.gons.add( Memories.gonsForBalance( _amount ) ),
            expiry: epoch.number.add( warmupPeriod ),
            lock: false
        });

        Memories.safeTransfer( address(warmupContract), _amount );
        emit LogStake(_recipient, _amount);
        return true;
    }

    /**
        @notice retrieve MEMO from warmup
        @param _recipient address
     */
    function claim ( address _recipient ) external {
        Claim memory info = warmupInfo[ _recipient ];
        if ( epoch.number >= info.expiry && info.expiry != 0 ) {
            delete warmupInfo[ _recipient ];
            uint256 amount = Memories.balanceForGons( info.gons );
            warmupContract.retrieve( _recipient,  amount);
            emit LogClaim(_recipient, amount);
        }
    }

    /**
        @notice forfeit MEMO in warmup and retrieve Time
     */
    function forfeit() external {
        Claim memory info = warmupInfo[ msg.sender ];
        delete warmupInfo[ msg.sender ];
        uint memoBalance = Memories.balanceForGons( info.gons );
        warmupContract.retrieve( address(this),  memoBalance);
        Time.safeTransfer( msg.sender, info.deposit);
        emit LogForfeit(msg.sender, memoBalance, info.deposit);
    }

    /**
        @notice prevent new deposits to address (protection from malicious activity)
     */
    function toggleDepositLock() external {
        warmupInfo[ msg.sender ].lock = !warmupInfo[ msg.sender ].lock;
        emit LogDepositLock(msg.sender, warmupInfo[ msg.sender ].lock);
    }

    /**
        @notice redeem MEMO for Time
        @param _amount uint
        @param _trigger bool
     */
    function unstake( uint _amount, bool _trigger ) external {
        if ( _trigger ) {
            rebase();
        }
        Memories.safeTransferFrom( msg.sender, address(this), _amount );
        Time.safeTransfer( msg.sender, _amount );
        emit LogUnstake(msg.sender, _amount);
    }

    /**
        @notice returns the MEMO index, which tracks rebase growth
        @return uint
     */
    function index() external view returns ( uint ) {
        return Memories.index();
    }

    /**
        @notice trigger rebase if epoch over
     */
    function rebase() public {
        if( epoch.endTime <= uint32(block.timestamp) ) {

```

```
            warmupContract.retrieve( _recipient,  amount);
            emit LogClaim(_recipient, amount);
        }
    }

    /**
        @notice forfeit MEMO in warmup and retrieve Time
     */
    function forfeit() external {
        Claim memory info = warmupInfo[ msg.sender ];
        delete warmupInfo[ msg.sender ];
        uint memoBalance = Memories.balanceForGons( info.gons );
        warmupContract.retrieve( address(this),  memoBalance);
        Time.safeTransfer( msg.sender, info.deposit);
        emit LogForfeit(msg.sender, memoBalance, info.deposit);
    }

    /**
        @notice prevent new deposits to address (protection from malicious activity)
     */
    function toggleDepositLock() external {
        warmupInfo[ msg.sender ].lock = !warmupInfo[ msg.sender ].lock;
        emit LogDepositLock(msg.sender, warmupInfo[ msg.sender ].lock);
    }

    /**
        @notice redeem MEMO for Time
        @param _amount uint
        @param _trigger bool
     */
    function unstake( uint _amount, bool _trigger ) external {
        if ( _trigger ) {
            rebase();
        }
        Memories.safeTransferFrom( msg.sender, address(this), _amount );
        Time.safeTransfer( msg.sender, _amount );
        emit LogUnstake(msg.sender, _amount);
    }

    /**
        @notice returns the MEMO index, which tracks rebase growth
        @return uint
     */
    function index() external view returns ( uint ) {
        return Memories.index();
    }

    /**
        @notice trigger rebase if epoch over
     */
    function rebase() public {
        if( epoch.endTime <= uint32(block.timestamp) ) {

            Memories.rebase( epoch.distribute, epoch.number );

            epoch.endTime = epoch.endTime.add32( epoch.length );
            epoch.number++;

            if ( address(distributor) != address(0) ) {
                distributor.distribute();
            }

            uint balance = contractBalance();
            uint staked = Memories.circulatingSupply();

            if( balance <= staked ) {
                epoch.distribute = 0;
            } else {
                epoch.distribute = balance.sub( staked );
            }
            emit LogRebase(epoch.distribute);
        }
    }

    /**
        @notice returns contract Time holdings, including bonuses provided
        @return uint
     */
    function contractBalance() public view returns ( uint ) {
        return Time.balanceOf( address(this) ).add( totalBonus );
    }

    enum CONTRACTS { DISTRIBUTOR, WARMUP }

    /**
        @notice sets the contract address for LP staking
        @param _contract address
     */
    function setContract( CONTRACTS _contract, address _address ) external onlyOwner {
        if( _contract == CONTRACTS.DISTRIBUTOR ) { // 0
            distributor = IDistributor(_address);
        } else if ( _contract == CONTRACTS.WARMUP ) { // 1
            require( address(warmupContract) == address( 0 ), "Warmup cannot be set more than once" );
            warmupContract = IWarmup(_address);
        }
        emit LogSetContract(_contract, _address);
    }

    /**
     * @notice set warmup period in epoch's numbers for new stakers
     * @param _warmupPeriod uint
     */
    function setWarmup( uint _warmupPeriod ) external onlyOwner {
        warmupPeriod = _warmupPeriod;
        emit LogWarmupPeriod(_warmupPeriod);
    }
}
```

```
            Memories.rebase( epoch.distribute, epoch.number );

            epoch.endTime = epoch.endTime.add32( epoch.length );
            epoch.number++;

            if ( address(distributor) != address(0) ) {
                distributor.distribute();
            }

            uint balance = contractBalance();
            uint staked = Memories.circulatingSupply();

            if( balance <= staked ) {
                epoch.distribute = 0;
            } else {
                epoch.distribute = balance.sub( staked );
            }
            emit LogRebase(epoch.distribute);
        }
    }

    /**
        @notice returns contract Time holdings, including bonuses provided
        @return uint
     */
    function contractBalance() public view returns ( uint ) {
        return Time.balanceOf( address(this) ).add( totalBonus );
    }

    enum CONTRACTS { DISTRIBUTOR, WARMUP }

    /**
        @notice sets the contract address for LP staking
        @param _contract address
     */
    function setContract( CONTRACTS _contract, address _address ) external onlyOwner {
        if( _contract == CONTRACTS.DISTRIBUTOR ) { // 0
            distributor = IDistributor(_address);
        } else if ( _contract == CONTRACTS.WARMUP ) { // 1
            require( address(warmupContract) == address( 0 ), "Warmup cannot be set more than once" );
            warmupContract = IWarmup(_address);
        }
        emit LogSetContract(_contract, _address);
    }

    /**
     * @notice set warmup period in epoch's numbers for new stakers
     * @param _warmupPeriod uint
     */
    function setWarmup( uint _warmupPeriod ) external onlyOwner {
        warmupPeriod = _warmupPeriod;
        emit LogWarmupPeriod(_warmupPeriod);
    }
}
```