Dutch Auction

Todo

- [ ] Add nonce to orders
- [ ] Add on-chain sorting

Table of content

Todo
High-level Design
Data Structure
    DutchAuction.sol
    StateMachine.sol
Requirements
    Further investigation
Gas Cost Analysis
Security Considerations
Vulnerabilities considerations
Testing
Extension - Hiding bids
Requirements
    Test cases
    Gas analysis
Reflection and discussion
Getting Started

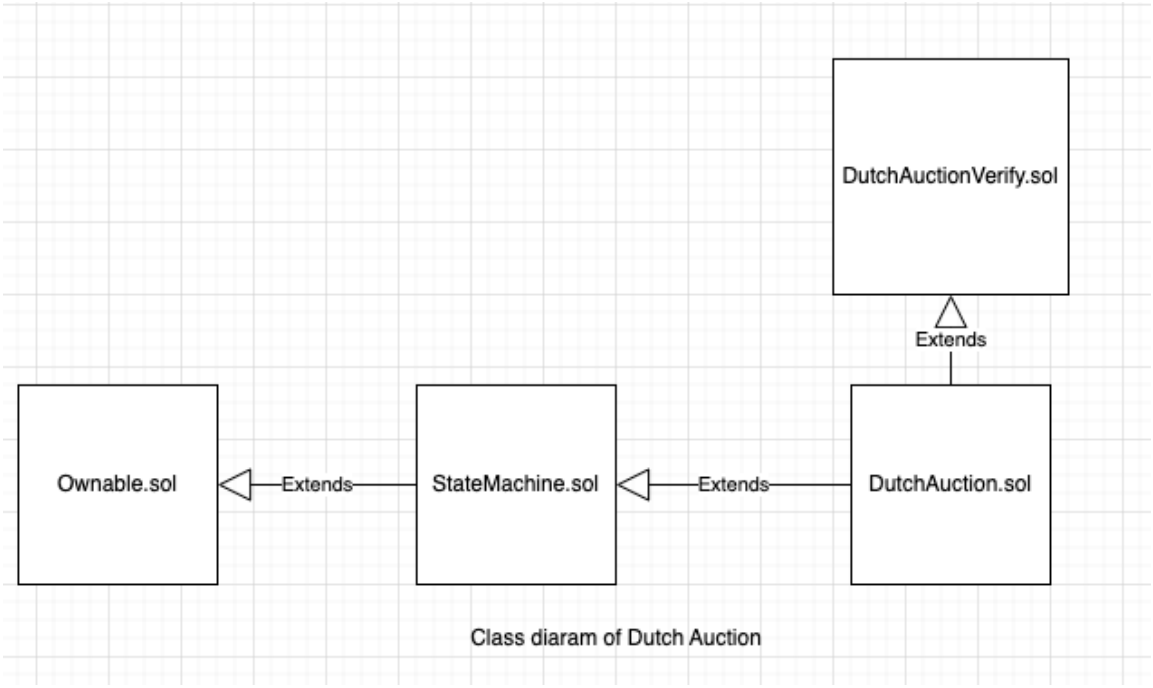
****See “Getting Started” to run the code.****

High-level Design

There are three smart contracts in this applications

- `Ownable.sol` Manages access control
- `StateMachine.sol` Manages auction state transition. It extends from `Ownable.sol`
- `DutchAuction.sol` Manages account balances and auction logic. It extends from `StateMachine.sol`
- `DutchAuctionVerify.sol` Manages signature verification logic

Further, I also included WBTC.sol and WETH.sol to mock ERC20 tokens to test simplicity.



Class diaram of Dutch Auction

Data Structure

DutchAuction.sol

Whitelisted Tokens

```
// Whitelisted tokens
mapping(bytes32 => address) public whitelistedTokens;
```

Our contract only allows ERC20 tokens that are in our whitelist. Only the contract owner can add whitelisted tokens.

Considerations:

1. Having a whitelist gives centralised decision power to the owner. However, without monitoring, the auction contract can grow out of control.

Wallet

```
// Token wallet
struct Wallet {
    uint256 liquid; // can be used to create bids/offers freely
    uint256 locked; // locked in bids and offers
}
```

Each type of token is mapped to a wallet struct that stores the liquid and locked balance.

When a user creates an offer, the amount of ERC20 token offered is **locked** and cannot be withdrawn as long as the offer is still ACTIVE. This is to ensure once an offer is matched by a

bid, the token transfer process can take place smoothly.

Only the liquid amount can be withdrawn.

Auction account

```
// ERC20 tokens balance
mapping(address => mapping(bytes32 => Wallet)) public accountBalances;

// ETH balance
mapping(address => Wallet) public ethBalances;
```

We use internal maps to track token balances. `accountBalances` tracks ERC20 token balances. `ethBalances` tracks ETH balances.

Note that, in our design, anyone can join the contract and deposit ETH and other whitelisted tokens.

Offer

```
enum OfferStatus {
  ACTIVE, // When an offer can be matched with a bid
  WITHDRAWN, // When an offer cannot be matched with a bid
  CLOSED // When an offer is fully matched with bids
}
struct Offer {
  uint256 id; // ID of this offer
  address created_by; // offer creator address
  bytes32 symbol; // When an offer is created, it is assumed that its been whitelisted
  uint256 quantity; // Token amount Fixed value
  uint256 remaining_quantity; // Used to track amount left in this offer
  uint256 offer_unit_price_eth; // Offer price in ETH
  uint256 creation_time; // Creation time
  OfferStatus status; // State of this offer
}

// List of offers
Offer[] public offers;

// Total offer count
uint256 public offer_count;
```

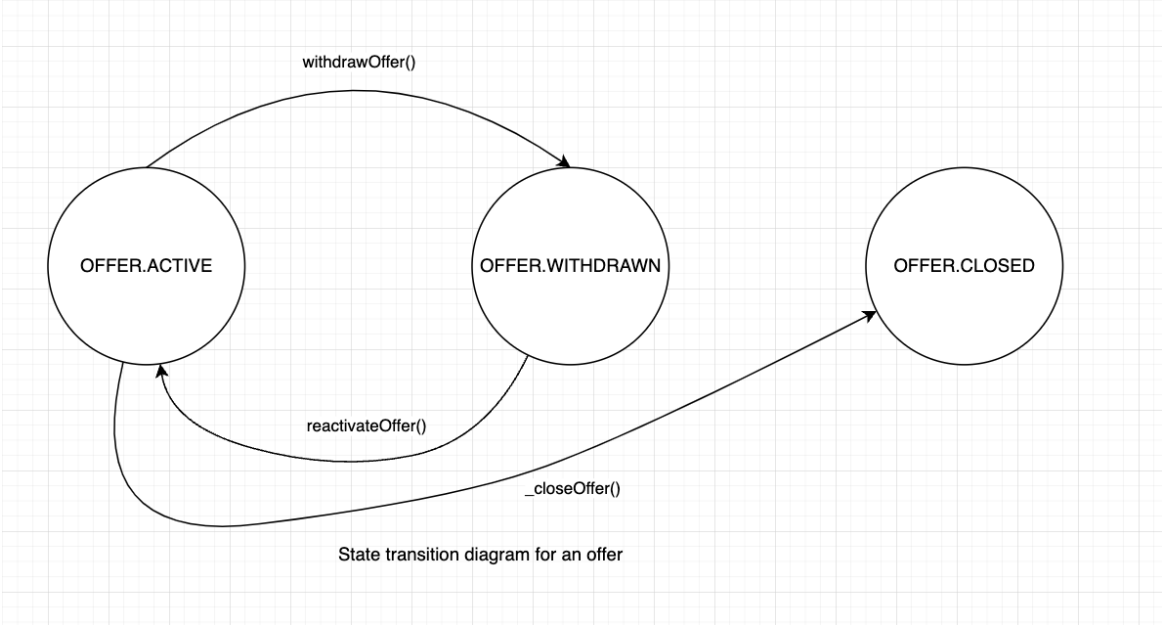
The most important attribute in an offer is `remaining_quantity.` An offer can be matched by several bids. `remaining_quantity` tracks the remaining amount of token in this offer so that later on, it can be matched by another bid.

An offer can take 3 states:

- ACTIVE: When an offer can be matched with a bid

- WITHDRAWN: When an offer cannot be matched with a bid
- CLOSED: When an offer is fully matched with bids

Below is the offer state transition diagram with the function that updates the transitions.



State transition diagram for an offer

The contract tracks a list of offers in `offers`, each offer can be accessed by its id as an array index.

Bid

```
// Bid states
enum BidStatusPrimary {
    HIDDEN, // When the content of the bid is hidden
    OPENED // When the bid is revealed
}

enum BidStatusSecondary {
    ACTIVE, // When the bid is able to match with offers
    WITHDRAWN, // When the bid is withdrawn by the bidder
    CLOSED // When the bid has been successfully matched with an offer
}

struct Bid {
    uint256 id;
    address created_by;
    bytes32 symbol;
    uint256 quantity;
    uint256 remaining_quantity;
```

```
    uint256 bid_unit_price_eth;
    uint256 creation_time;
    BidStatusPrimary primary_status;
    BidStatusSecondary secondary_status;
}

Bid[] public bids;
uint256 public bid_count;
```

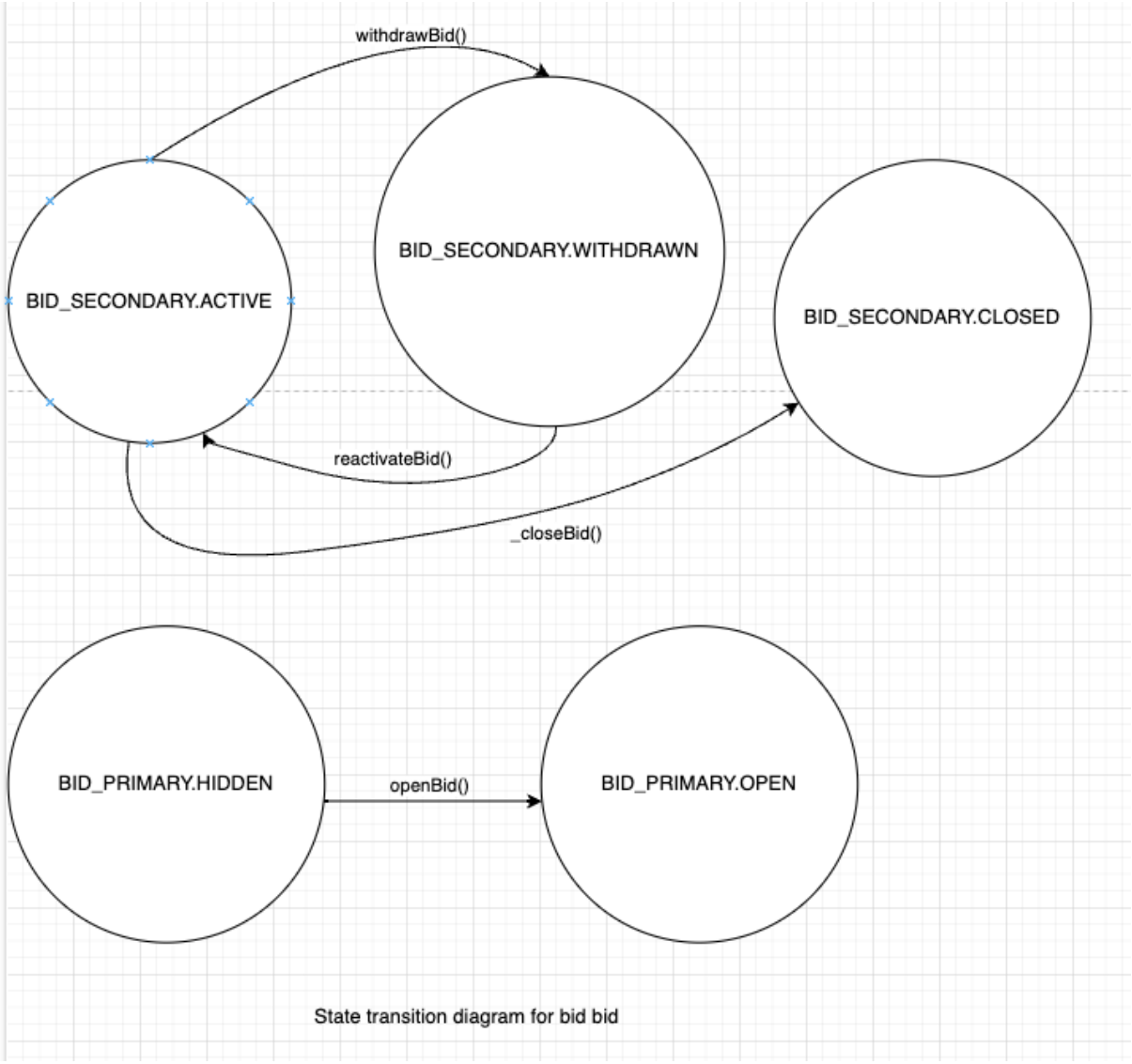
The data structure of a bid is similar to an offer. A major difference is that it has two states at any given moment - a primary state and a second state.

The primary state indicates if the bid can be matched by an offer. If it is HIDDEN, it cannot join a matching round and cannot be seen by other users.

The secondary state indicates the active status of the bid similar to an offer.

The contract tracks a list of bids in `bids`, each bid can be accessed by its id as an array index.

Below is the bid state transition diagram with the function that updates the transitions.



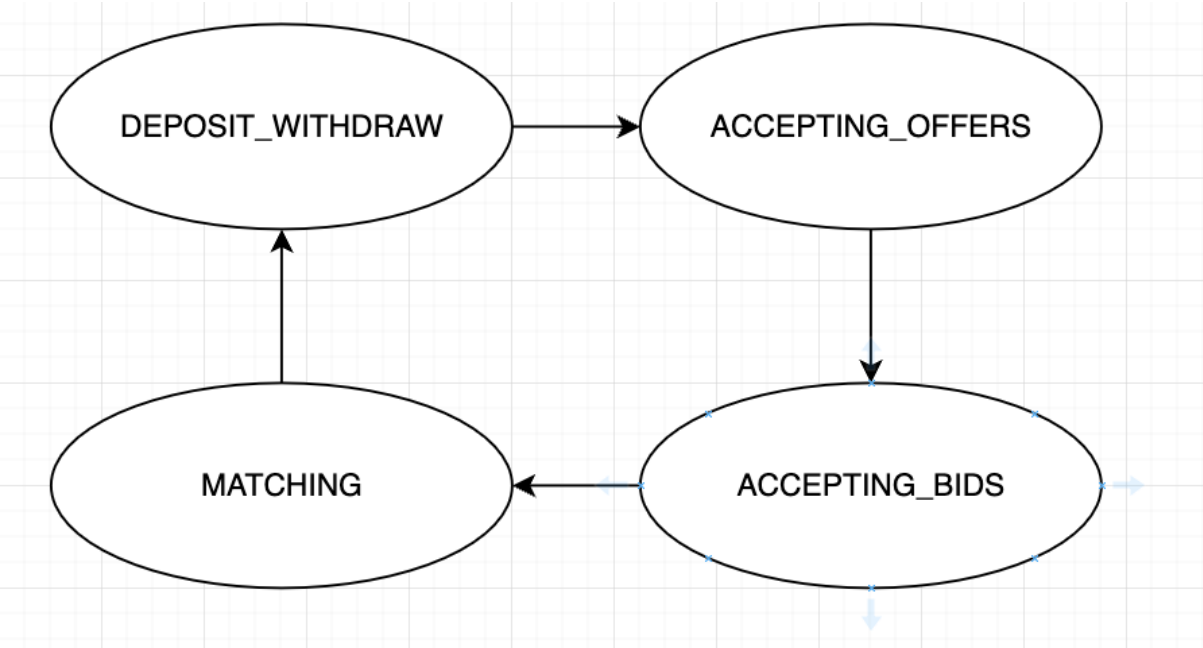
StateMachine.sol

`StateMachine.sol` is used to track the stage we are currently at.

```
enum Stages {
    DEPOSIT_WITHDRAW, // Open for deposit and withdraw
    ACCEPTING_OFFERS, // Accepting new offers and update offers
    ACCEPTING_BIDS, // Open bids, place bids, update bids
    MATCHING // Match offers with bids
}
```

According to the requirement, each stage should last roughly 5 minutes. This may not always be the case because the new state transfer are triggered by a valid transaction, not by time. More on this in the next section.

Below is the state transition diagram



Auction state transaction diragram

Requirements

1. *Buyers and Sellers should be able to create accounts at the smart contract, containing an amount of ETH, and a set of ERC-20 tokens of different kinds. (Token kinds are identified by the address of the smart contract from which they were issued.)*
2. *ERC-20 tokens held in the market should be under the control of the DutchMarket contract, so that they cannot be sold elsewhere.*
3. *Buyers and Sellers should be able to deposit and withdraw both tokens and ETH currency from their accounts.*

To address 1-3:

Buyer can call the following functions to deposit ETH and ERC20 tokens:

1. `depositETH()` to deposit ETH
2. `withdrawETH(withdraw_amount)` to withdraw ETH
3. `depositTokens(symbol, amount)` to deposit ERC20 tokens. Note that only whitelisted tokens can be deposited
4. `withdrawTokens(symbol, amount)` to withdraw ERC20 tokens.

4. *Sellers should be able to offer to sell a block of N tokens of the same kind from their account, at a specified price.*

To address 4

After user deposited some amount of ERC20 tokens, they can call `createOffer(symbol, quantity, unit_price)` to create an offer. Note that when an offer is created, they will not be able to withdraw the amount of token listed in that offer. They can liquidate the tokens by withdrawing the offer.

5. *Sellers should be able to reduce the price on an outstanding offer, but not increase the price.*

With an active offer, the owner can call `decreaseOfferPrice(offer_id, new_unit_price)` to update the price. Note that it will fail if the new price is equal or higher than the current price.

To address 5

The deposited tokens are stored in the contract. The account details and balance are stored in `accoundBalances` and `ethBalances` mappings.

6. *Sellers should be able to withdraw an offer to sell.*

To address 6

The offer creator can withdraw an ACTIVE offer by calling `withdrawOffer(offer_id).`

Additional requirement: After withdrawing offer, the fund locked in the offer is released.

7. *Buyers should be able to make blinded bids to buy tokens. A bid states the token type for the purchase, the number of tokens to be bought, as well as a maximum price. These details of the bid should not be deducible by anyone who is monitoring the blockchain. However, the cryptographic address of the bidder may be revealed (see the next section, which addresses hiding of the bidder identity.)*

To address 7

Each bid has two types of states - `BidPrimaryStatus` which tracks if a bid is HIDDEN or OPENED. A HIDDEN bid cannot be matched by an offer where as an OPENED bid can. And `BidSecondaryStatus`, which tracks a bid's active status similar to an offer.

8. *When a seller reduces the price to an amount that a buyer is prepared to pay, the buyer makes the purchase by opening a matching blinded bid. This reveals the token type, number of tokens and maximum price.*

9. *Once a buy offer has been opened, it remains visibly open.*

To address 8 and 9

During ACCEPTING_BID, a bidder can call `openBid(bid_id)` to change a bid's primary state to `OPENED` so that it can join the next matching round. Note that once a bid is OPENED, it cannot be reverted to HIDDEN.

10. *A buyer may withdraw an open or closed offer.*

To address 10

Similar to withdrawing an offer, a bidder can withdraw a bid by calling `withdrawBid(bid_id)` when the primary state is HIDDEN or OPENED.

11. *The market operates in a repeated sequence of modes: Deposit/Withdraw, Offer, Bid Opening and Matching…..*

To address 11

I implemented multiple modifiers in `StateMachine.sol` to check block timestamp of each transaction to track time. These modifiers are put in front of all public/external functions to ensure functions specific to each state will run independently.

Additionally, I added an `ownerOnly` function `godModeSetState()` to allow the owner to set the current state for the ease of testing and contract management.

12. *For each matching pair, the appropriate number of tokens k is transferred from the seller's account to the buyer's account, and k times the offe rprice is transferred from the buyer's account to the seller's account. (The service does not charge a market fee to buyers and sellers.)*

To address 12

After a bid is matched to an offer, three private functions `_transferEthFromBuyerToSeller`, `_transferTokenFromSellerToBuyer`, and `_refundExcessEth`, are called to transfer eth from bidder to seller, transfer ERC20 from seller to bidder, and refund excess ETH locked in a bid to the bidder if the bid price is higher than offer price.

13. *Negative account balances are prohibited.*

To address 13

Multiple `require` are used to make sure before withdraws, account balances are positive and cannot be over-withdrawn.

e.g.

```
require(_amount > 0, "Can only withdraw positive integer amount");
require(
  accountBalances[msg.sender][_symbol].liquid >= _amount,
  "Insufficient liquid balance"
);
```

Additional requirements addressed

- An offer cannot be re-activated if the user does not have enough funds to fulfil a match.
- After MATCHING, the state returns to DEPOSIT_WITHDRAW
- Token quantity, Bid price and offer price must be positive integers.
- Note that there is a fallback function to capture funds
- At the beginning of the auction, the state is set to `DEPOSIT_WITHDRAW`

Considerations

- Since the state changes are triggered by new and valid transactions, there is a chance that the contract could go stale if no valid transaction are submitted. To address this, I added a `probeState()` method which can be used to check if a valid state transition is allowed.
- Who pays to call `matchOffers()`?

Further investigation

- Current implementation keeps closed bids and offers. It is costly to keep the data on chain as the list grows. Should we remove closed bids and offers to prevent higher running cost.
- `findCheapestOffer(symbol)` find the cheapest offer in the auction given a symbol. It does this by iterating through all offers. Not sure if this is costly yet. It's a view function but not sure how costly it can be as more offers gets appended into the list.
- Add a mapping to track offers linked to individual bids (hidden and opened) and offers.

Gas Cost Analysis

From etherscani.io/gastracker we consider average gas price of the 35 transactions happened on 01/04/2023 9AM, which is 33 GWEI to estimate gas cost of our functions.

Further, we use `web3.js` to help us estimate the cost of calling each public function.

As of EIP-1559 the gas fee is calculated as follows: **((base fee + priority fee) x units of gas used)**

we take base fee = 33 GWEI

One Ethereum in AUD on 01/04/2023: $AUD2724.06

Function	State	Gas units used	Total fee (Gwei)	In AUD ($)
`addToken(bytes32 symbol, address tokenAddress)`	Any	47027	1551891	4.2274442
`probeStage()`	Any	32503	1072599	2.92182403
`depositTokens(uint256 amount, bytes32 symbol)`	DEPOSIT_WITHDRAW	94267	3110811	8.47403581
`withdrawETH( uint256 _amount )`	DEPOSIT_WITHDRAW	35706	1178298	3.20975445
`withdrawTokens( uint256 _amount, bytes32 _symbol )`	DEPOSIT_WITHDRAW	51151	1687983	4.59816697
`createOffer( bytes32 _symbol, uint256 _quantity, uint256 _offer_unit_price_eth )`	ACCEPTING_OFFERS	238849	7882017	21.4710872
`decreaseOfferPrice( uint256 _offer_id, uint256 _new_unit_price )`	ACCEPTING_OFFERS	40921	1350393	3.67855156
`reactivateOffer( uint256 _offer_id )`	ACCEPTING_OFFERS	69024	2277792	6.20484208
`function withdrawOffer( uint256 _offer_id )`	ACCEPTING_OFFERS	66142	2182686	5.94576763
`createBid( bytes32 _symbol, uint256 _quantity, uint256 _bid_unit_price_eth )`	ACCEPTING_BIDS	240921	7950393	21.6573476
`withdrawBid( uint256 _bid_id )`	ACCEPTING_BIDS	66313	2188329	5.9611395
`reactivateBid(uint256 _bid_id)`	ACCEPTING_BIDS	69449	2291817	6.24304702
`openBid(uint256 _bid_i)`	ACCEPTING_BIDS	56006	1848198	5.03460224
`matchOffers()`	MATCHING	123202	4065666	11.0751181

Analysis:

From the bidder’s perspective:

1. It is costly to create a bid. Therefore the bidder should consider bid profitability before placing or updating a bid.

From the seller’s perspective:

1. Similar to the bidders, the most costly operation for a seller is to create an offer. Therefore the seller should consider a good price first before creating an offer to avoid reducing the price later on which incurs additional costs.

Future considerations:

1. Use external contracts to store data: Store data in external contracts rather than in the current contract. This can be more cost-effective since external contracts can share storage and reduce the overall storage usage.
2. Use smaller data types: In the current implementation unsigned integers are all uint256, which is more costly to store. In the future, the contract can be optimised with smaller data type. For example, using uint8 instead of uint256 can reduce the amount of storage required for each data item.
3. Purge old data: Delete offer and bid data that is no longer needed in the smart contract to free up storage space.

Security Considerations

The users of a smart contract should note the following security considerations before using a smart contract.

1. Smart contract vulnerabilities: Smart contracts can be vulnerable to attacks, such as reentrancy attacks, integer overflows, and underflows, etc. It is crucial to understand the common smart contract vulnerabilities and design the contract in a way that mitigates the risks associated with these vulnerabilities.
2. Transaction order dependency: Smart contracts can be affected by the order in which transactions are executed, which can create race conditions that can be exploited by attackers. In our case, since our contract is time dependent (or block dependent), network delay can cause a transaction to revert due to time misalignment in the state transition process.
3. External contract dependencies: Smart contracts that rely on external contracts or services are more susceptible to attacks since they can be exploited if the external contract or service is compromised. It's important to perform due diligence when choosing external contracts and services and ensure that they are secure and reliable.

To addressed these considerations:

1. Users should perform due diligence on the contracts to see if the smart contracts utilises established frameworks such as OpenZeppelin, Truffle, etc. Using these framework can reduce the likelihood of introducing vulnerabilities.
2. Audit access control mechanism to see who can access which methods. This way, the user will understand how other people could potentially take control of the contract.

3. Use a testnet to test and try the contract before using it on the main-net to understand behaviours of it.

Vulnerabilities considerations

Reentrancy Attack of ETH

To prevent reentrancy attacks, I implemented the “withdraw pattern”. The pattern involves separating the balance of a contract from the contract's internal state, and only allowing authorized parties to withdraw funds from the contract.

In this pattern, the contract maintains a mapping (`mapping(address => Wallet) public ethBalances;`) of the ETH balances of individual users or contracts. When a user or contract wants to withdraw funds, they call a "withdrawETH" function in the contract, passing in the amount they want to withdraw. The contract then updates its internal balance and the user's external balance and sends the requested funds to the user or contract address.

If a malicious contract were to try to exploit re-entrancy by calling the "withdraw" function repeatedly, the contract would check the user's balance and update the balance mapping before making any external calls to transfer tokens. Therefore, if the malicious contract tries to call the "withdraw" function again during the same transaction, the balance mapping would have already been updated and the balance check would fail, preventing any further transfers of tokens.

In the below implementation, only the message sender can withdraw their funds.

```
/**
 * @notice Withdraw ETH from contract, must have enough balance.
 * @dev _amount > 0
 * @param   _amount  Amount to withdraw > 0.
 */
  function withdrawETH(
      uint256 _amount
  ) external payable timedTransitions atStage(Stages.DEPOSIT_WITHDRAW) {

      // account has enough money
      require(
          ethBalances[msg.sender].liquid >= _amount,
          "Insufficient balance"
      );

      ethBalances[msg.sender].liquid -= _amount;

      (bool success, ) = msg.sender.call{value: _amount}("");
      require(success, "Failed to send funds");

      emit withdrawEth(msg.sender, _amount);
  }
```

Reentrancy Attack of ERC20 Token

Similarly, I applied the “withdraw pattern” to ERC20 token withdraw. When a user wants to withdraw tokens, they call the "withdrawTokens" function and the contract transfers the requested tokens to the user's address using the "transfer" function of the ERC20 token contract.

The token balance for each user is maintained in `mapping(address => mapping(bytes32 => Wallet)) public accountBalances;`

```
    /**
     * @notice  Withdraw ERC20 tokens.
     * @param   _amount  Token amount.
     * @param   _symbol  Token symbol.
     */
    function withdrawTokens(
        uint256 _amount,
        bytes32 _symbol
    )
        public
        timedTransitions
        atStage(Stages.DEPOSIT_WITHDRAW)
        tokenInWhitelist(_symbol)
    {
        require(_amount > 0, "Can only withdraw positive integer amount");
        require(
            accountBalances[msg.sender][_symbol].liquid >= _amount,
            "Insufficient liquid balance"
        );
        address tokenAddress = whitelistedTokens[_symbol];

        IERC20 token = IERC20(tokenAddress);
        accountBalances[msg.sender][_symbol].liquid -= _amount;

        require(
            token.transfer(msg.sender, _amount),
            "Unable to transfer token"
        );
        emit withdrawToken(msg.sender, _symbol, _amount);
    }
```

Access Control

To prevent random account from accessing methods in the contract, my Dutch Auction contract inheriated from `Openzeppelin`'s Ownable.sol contract.

By default, the owner of the auction is the creator of the contract. Ownable.sol allows me to `transferOwnership` and `renounceOwnership`.

Testing

A total of 55 unit and integration tests using the Truffle development suite were written to Testing files can be located in `$ test` directory.

To run existing tests:

- Start a `ganache` instance by running `$ ganache`
- In a separate termina run `$ truffle test` in the project root directory

To run individual test files:

- Run `$ truffle test ./test/<file name>`

Test cases and their description are listed below.

```
Contract: DutchAuction: Bid operations
    Bid creation
      ✔ Should succeed: should have no bids at the start
      ✔ Should create bid
      ✔ Should fail to create bid: Token not in whitelist
      ✔ Should fail to create bid: insufficient ETH liquid balance to purchase
    Bid update
      ✔ Should fail: update unexisted bid
      ✔ Should succeed: withdraw bid
      ✔ Should fail: others try to withdraw your bid
      ✔ Should fail: withdraw a WITHDRAWN bid
      ✔ Should succeed: reActivate bid
      ✔ Should fail: reActivate bid and ACTIVE bid
      ✔ Should succeed: Open bid
      ✔ Should fail: Open an alreay opened bid
      ✔ Should fail: Cannot open a WITHDRAWN bid

Contract: DuctchAuction: Deposit tokens
    ✔ Should succeed: wallet 1 should have 500 tokens
    ✔ Should succeed: deposit ERC20 tokens
    ✔ should not deposit ERC20 tokens if the depositor has insufficient balance
    ✔ should not deposit non-whitelisted ERC20 tokens
    ✔ Should succeed: Wallet1 deposit 1 wei
    ✔ Should succeed: Wallet1 withdraw 1 wei
    ✔ Should fail: Wallet1 over withdraw

Contract: DutchAuction: Match Operations
    find cheapest offer
      ✔ Should succeed: find cheap offer
      ✔ Should fail: no cheap offer
    match offer
      ✔ Should succeed: simple matching
      ✔ Should succeed: simple matching with offer excess
      ✔ Should succeed: Multiple bids with excess funds transfered

Contract: DutchAuction: Offer Operations
    Create offers
      ✔ should no offers
      ✔ should succeed: create an offer
      ✔ Should fail: create offer with token not in whitelist
      ✔ should fail: insufficient balance to offer
    Update offer
```

```
      ✔ Should succeed: decrease offer price
      ✔ Should fail: decrease offer price when offer does not exist
      ✔ Should fail: new price is higher than current price
      ✔ Should succeed: change offer state to WITHDRAWN
      ✔ Should fail: change offer state to WITHDRAWN when it is already WITHDRAWN
      ✔ Should succeed: Offer state changed from WITHDRAWN to ACTIVE
      ✔ should fail: to re-activate offer due to lack of liquid balance
      ✔ Should fail: Offer state change to ACTIVE when it is already ACTIVE
      ✔ Should fail: Others try to update your offer

Contract: DutchAuction: Withdraw token
    ✔ Should succeed: check initial user contract balance
    ✔ Should succeed: withdraw WBTC
    ✔ should fail: withdraw due to insufficient balance
    ✔ should fail: withdraw tokens that do not exist

Contract: StateMachine: State transition tests
    Stage tests: Deposit and Withdraw
      ✔ Should fail: perform bidding operations out of turn
      ✔ Should fail: perform offer operations out of turn
      ✔ Should fail: perform matching operations out of turn
    Stage tests: Offer
      ✔ Should fail: perform deposit and withdraw operations out of turn
      ✔ Should fail: perform bidding operations out of turn
      ✔ Should fail: perform matching operations out of turn
    Stage tests: Bid opening
      ✔ Should fail: perform deposit and withdraw operations out of turn
      ✔ Should fail: perform offer operations out of turn
      ✔ Should fail: perform matching operations out of turn
    Stage tests: Matching
      ✔ Should succeed: Matching and state change right after
      ✔ Should fail: perform deposit and withdraw operations out of turn
      ✔ Should fail: perform offer operations out of turn
      ✔ Should fail: perform Bid operations out of turn
```

Extension - Hiding bids

Working implementation of the extension section is placed under a separate directory in `dutch_extension`.

General idea of implementation:

- To hid the real bidder information and the bid content, two new fields are added in the bid structure:
  - `bid_owner`: the actual owner address, it is set to null when the bid is hidden
  - `bid_hash`: the has of the bid content which stores `symbol`, `quantity`, and `unit_price_eth`
    - this has is created off-chain to preserve privacy.
  - We used `web3.js` to assist us in creating the hash

    - The reason to why we are still creating a bid struct is to keep the bidding creating order so that the firstly placed order will be processed sequentially.
- Unlike the general implementation, when creating a bid now, the bid owner is not required to have enough balance in its contract account since this bid is hidden and we cannot perform checks.
- Visible fields in the bid struct will include: bid id, symbol, primary_status, secondary_status. (see below)
- When the user calls `openBid(id)`, the hidden fields are populated by opening the hash. At the same time, the contract will check if the user has enough balance in its account to have a valid bid. If they don't have enough ETH, then it cannot be opened.
- The bid primary status is changed to `OPENED` after calling `openBid(id)`.
- Once the bid is opened, the rest of the auction operations will stay the same.

New data structures:

```
struct Bid {
  uint256 id;
  address created_by;
  address bid_owner; // new: the actual owner of the bid. At the start is set to null address
  bytes32 symbol;
  uint256 quantity;
  uint256 remaining_quantity;
  uint256 bid_unit_price_eth;
  bytes32 signature; // new: signed hash of the bidding content
  uint256 creation_time;
  BidStatusPrimary primary_status;
  BidStatusSecondary secondary_status;
}
```

`Bid` introduced two new fields

- `address bid_owner;` stores the actual owner address. It is set to null at before the bid is opened.
- `bytes32 signature`: stores the signed hash of the bid content

Requirements

The **flow** of sending and opening a hidden bid is divided into two parts - off-chain and on-chain.

Off-chain

- Bidder uses two accounts. The first one, *A*, is their own, the second one, *B,* is the one sending the hidden bid to the auction contract.
- *B* creates a hidden bid off-chain
  - The bid content (symbol, token quantity, and unit price) is hashed and signed using A's private key.
  - B then sends the hashed/signed message to the contract to create a new bid.

On-chain

- To open a bid, *A* calls `openBid(bid_owner, symbol, quantity, unit_price_eth)` which performs the following operations
  - checks if the input arguments are valid
  - recreates a hash of `[bid_owner, symbol, quantity, unit_price_eth]` and check if the equals to the bid on-chain
  - verifies if the user opening the bid is in fact the bid owner, who signed the hash.
    - Retrieve the message signer by using `ercrecover()` and verify is the user opening the bid is in fact the user signing the original transaction.
  - Once verified, the hash will be unpacked and the hidden value will be filled.
  - The primary state of the bid will be updated to open.

After a bid state is changed to opened, the rest of the operations follow as before.

Potential issues:

- Following this process, calling `createBid()` does not guarantee that the information in the bid will be valid. This can create the issue that an invalid bid takes the space on chain and persists since there is no method of removing bids currently.

Test cases

New tests cases:

```
Contract: DuctchAuction: Hidden bids
Dutch Auction: Bid creation
  ✔ Should Succeed: Hidden bid creation (create from own account)
  ✔ Should Succeed: Hidden bid creation (create from another account)
Dutch Auction: Bid opening
  ✔ Should Succeed: open a bid
  ✔ Should fail: open a bid that is not yours
  ✔ Should fail: bid signature mismatch
  ✔ Should fail: Open bid with insufficient balance
  ✔ Should fail: Token not in whitelist
```

Gas analysis

Other operations except for Bid operations remained mostly unchanged in terms of gas consumed. However, `openBid()` became a lot more expensive because of the signature verification operations. Before adding signature verification the gas units used is 56006, after the verification logic is added, the number of gas units required became 209213, nearly quadrupling the cost, making it similar to creating a bid.

Reflection and discussion

There are many benefits of using Ethereum as a platform for applications such as a dutch auction.

First, transactions made on Ethereum are transparent. In our case, we have full transparency to all offers and bids placed in the past. It is difficult to hide information, other than hidden bids.

Secondly, smart contracts allow for automation of certain aspects of the auction process such as distributing of funds and assets. This reduces manual intervention and makes the process more efficient.

Third, Ethereum is immutable, meaning malicious users cannot go back in time and modify or manipulate historical records. The immutability aspect of Ethereum makes it more secure.

However, there are some downsides to it.

Firstly, the cost of running a smart contract can be costly due to high transaction storage fees. As shown in previous section on cost, at the moment, it costs $9 AUD to deposit some ERC20 tokens, $20 AUD to create an offer or bid, and $20 to open a bid. This is very expensive for auctions with lower value, making it inaccessible to the general public.

Secondly, deploying a smart contract requires high level of technical expertise. Since the process involves exchange of assets, the code requires extensive testing and auditing to prevent hacking. There are many examples where bugs in a smart contract cost users millions of dollars.

Thirdly, The Ethereum blockchain has a limited capacity for processing transactions, which can result in delays and higher fees during periods of high activity. This could impact the user experience of the auction process.

Getting Started

Directory structure

```
Assignment/
├─ dutch/
│  ├─ contracts
│  ├─ test/
│  ├─ truffle-config.js
│  ├─ package.json
├─ dutch_extension/
│  ├─ contracts
│  ├─ test/
│  ├─ truffle-config.js
│  ├─ package.json
```

For general implementation

run `$ cd dutch`

For extension implementation

run `$ cd dutch_extension`

System requirement

- install solidity: `$ npm install solc --global`
- install ganache: `$ npm install ganache --global`
- install truffle: `$ npm install truffle --global`

Dependencies:

run `$ npm install` in project root to install dependencies

```
{
  "dependencies": {
    "@openzeppelin/contracts": "^4.8.2",
    "bn.js": "^5.2.1",
    "ganache-time-traveler": "^1.0.16",
    "truffle-assertions": "^0.9.2",
    "web3": "^1.9.0"
  }
}
```

To run existing tests:

- Start a `ganache` instance by running `$ ganache`
- In a separate termina run `$ truffle test` in the project root directory

To run individual test files:

- Run `$ truffle test ./test/<file name>`