UNSW COMP6451
Assignment 2 : Ethereum Programming
(ERC-20 Token Dutch Auction Market)

Total Marks: 35

Due Date: 5pm, March 31, 2023

Background

A variety of schemes are used to sell goods in such a way as to yield the highest profit for the seller. English Auctions, used in Australia for selling real estate, are probably the most familiar to you: bidders take turns to offer increasingly high prices, until one bidder remains who has offered the highest price, and no other bidder is prepared to offer any higher price. The auction is then declared closed by the auctioneer, and the highest bidder is declared the successful buyer, at a price equal to their highest bid.

This approach has some disadvantages from the point of view of an unsuccessful bidder. During the course of the auction, they were required to reveal the highest price that they were prepared to pay. Other sellers wishing to sell similar items can then take advantage of this fact and refuse to sell for less than this price, depriving the purchaser of the possibility of a purchase at a lower price, even when there is no competition from other bidders. It would be preferable if the winning bidder for an auction were determined in a way that does not reveal the bids of the losing bidders. In part 1 of this assignment we develop a way to achieve this. Even better would be to also hide the *identity* of the losing bidders. This is addressed in a stretch goal in part 2 of the assignment.

Hiding the bids of losing bidders can be achieved through use of a *Dutch Auction.* Such an auction swaps the roles of the buyers and seller, and proceeds by successive *decreases* of the price, until a buyer is found who is prepared to pay that price. That is, the seller first states a high price, and asks the buyers for an offer to buy at that price. If no buyer states that they are prepared to pay that price, the seller reduces the price, and again asks for an offer. This process repeats, with the price decreasing over time, until a buyer is found.

In this assignment, we will apply this approach to develop a smart contract for a market where ERC-20 tokens can be bought and sold. Compared to the sale of a single item, we have some additional requirements. Consider a situation where a seller wishes to sell some number N of ERC-20 tokens of the same type. Suppose there are $n \geq 1$ bidders, all offering to buy $b_1, \ldots, b_n$ tokens at the current seller price p per token. Bidder i may receive a number $x \leq b_i$ tokens from this seller. Bidder i will pay price p per token when receiving x tokens from this seller, for total cost $x \times p$.

- If $\sum_{i=1 \ldots n} b_i \leq N$, each buyer i should receive exactly b_i tokens, leaving $N - \sum_{i=1 \ldots n} b_i$ remaining to be sold. The seller has the option of further reducing the price to sell these tokens if they wish.
- If $\sum_{i=1 \ldots n} b_i > N$, then there is contention for the tokens being sold, and we need a rule to determine how many tokens each buyer receives. We will take the attitude that the buyer who has been waiting to buy the longest will have their order filled first. For that, we need a way to determine which buyer that is, without revealing the price that they are prepared to pay too early. We resolve that by using bid blinding. We maintain a counter *bid-number* initially 0. When a buyer submits a buy order, it is assigned the current bid-number, and *bid-number* is incremented. The bid itself is *blinded*: it should not reveal the number of tokens that the buyer is offering to buy, nor the price they are prepared to pay for each token. When the seller's price reaches the price that the buyer is prepared to pay, the buyer *opens*, or *reveals* their bid, making the number of tokens and the price public. It should not be possible for the buyer to cheat by revealing a number of tokens or a price that differs from the numbers used in their initial blinded bid.

In the context of a market service, that provides matching between multiple sellers and buyers, there is a similar issue when multiple sellers are offering to sell the same kind of tokens, but at potentially different prices. ERC-20 tokens in the same asset (i.e., issued from the same smart contract) are *fungible*, so the buyers will not care which seller's tokens they receive. When matching buy bids and sale offers, the bids should be matched against the lowest price sale offers first. For sale offers at the same price, the older sale offers should be used first to supply the buyers. To enable this, when a block of tokens is first offered for sale, it is associated with an *offer-number*.

Note that where an offer to sell at price p is matched with a bid to buy at price $q \geq p$, the buyer pays the lower price p. A bid to buy at a price q should *not* be matched with an offer to sell at a price $p > q$: such an open bid is left in the market for possible matching in later rounds.

For example, suppose we have the following sale offers and opened buy bids, all for the same token type, in a given matching round.

- Seller 1 offers 5 tokens at price 3 ETH each, with offer-number 1
- Seller 2 offers 10 tokens at price 2 ETH each, with offer-number 2

- Buyer 3 bids for 5 tokens at price 3 ETH, with bid-number 1
- Buyer 4 bids for 20 tokens at price 2 ETH ,with bid-number 2

Then

- Buyer 3, who has the oldest bid, is handled first. Their bid is matched to Seller 2's lower priced offer, and Buyer 3 receives 5 tokens at price 2 ETH each. This leaves 5 tokens in Seller 2's offer.
- Buyer 4 is handled next. They receive 5 tokens at price 2 ETH from Seller 2's offer, and 5 tokens at price 3 ETH from Seller 1's offer. That is, they receive 10 tokens at total cost of 10 + 15 = 25 ETH.
- Buyer 4's bid was not completely filled, and remains open as a bid for 10 tokens at price 2 ETH, unless Buyer 4 elects to withdraw this offer during the next Bid Opening round.

Part 1 (25 marks)

Develop a DutchMarket smart contract in Solidity implementing a market for ERC-20 tokens that implements the above ideas. More precisely, the contract should have the following functionality:

- 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.)
- ERC-20 tokens held in the market should be under the control of the DutchMarket contract, so that they cannot be sold elsewhere.
- Buyers and Sellers should be able to deposit and withdraw both tokens and ETH currency from their accounts.
- Sellers should be able to offer to sell a block of N tokens of the same kind from their account, at a specified price.
- Sellers should be able to reduce the price on an outstanding offer, but not increase the price.
- Sellers should be able to withdraw an offer to sell.
- 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.)

- 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.
- Once a buy offer has been opened, it remains visibly open.
- A buyer may withdraw an open or closed offer.
- The market operates in a repeated sequence of modes: Deposit/Withdraw, Offer, Bid Opening and Matching, which restrict the operations that participants may perform while the system is the mode. Each mode applies for a period of 5 minutes, at the end of which, the market switches to the next mode in the sequence (Matching is followed by Deposit/Withdraw). The operations permitted in each mode are as follows:
  - **Deposit/Withdraw:** buyers and sellers may deposit and withdraw from their accounts
  - **Offer:** Sellers may make new offers to sell, withdraw offers, and reduce the price on existing offers to sell.
  - **Bid Opening:** Buyers may place new blinded bids to buy. Buyers may also open blinded bids. They may also withdraw open or blinded bids.
  - **Matching:** Outstanding offers to sell are matched with opened bids to buy, according to the priority rules described in Section 1.
- 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 offer price is transferred from the buyer's account to the seller's account. (The service does not charge a market fee to buyers and sellers.)
- Negative account balances are prohibited. Where execution of a match would create a negative account balance in either the buyer or seller's account, that match should not be executed. To the largest extent possible, the market design should prevent the possibility that such improper matches will occur.
- As much as possible, the implementation of the smart contract should minimize the gas cost that users (buyers and sellers) should be required to pay for their transactions.

The project has a strict deadline, so you should attempt the Part 1 specification first. For additional marks once you have completed this, attempt the stretch goal.

Part 2 - Stretch Goal (10 Marks)

As noted above, blinded bids still reveal the identity of the bidder, at least by way of their Ethereum address, because a user will open an account with that address, and then submit bid transactions to the market smart contract that are signed with the key for that address. This may be an issue for bidder privacy, and other bidders may prefer not to bid when they see that there is a competing bidder with a large cash balance, or who has a reputation of placing very high bids.

As a stretch goal, modify your solution to allow the bidder identity to be obscured by the following technique. Instead of placing blinded bids using transactions signed by the same address A under which they have opened an account at the market, users may submit blinded bids using transactions signed by another address B. These blinded bids state, as usual, the token type, number of tokens and a price. However, in addition, the bid states the address of the real bidder A. All of this information (except B) should be hidden, until the bid is opened. At the time of opening, the usual bid information, as well as A's address, should be revealed, so that matching of the bid will affect A's account in the usual way.

Note that the information revealed at the time of opening such a bid should prove that it really was A's intention to place the bid transaction signed by B. It should *not* be possible for any user other than the real bidder A to use this mechanism to cause changes to A's account. For this, the information revealed by bid opening should be a message, signed using address A, that states the usual bid information. The market smart contract should verify this signature before accepting the bid as valid and assigning it to user A. (You might like to think of the mechanism as bids being like sealed envelopes containing signed letters.)

Hint: The Ethereum function **ecrecover** and frameworks such as Web3.py or Web3.js are relevant to this part. You may also wish to investigate proposals such as EIP-721 and EIP-2612.

Deliverables

Submit the following. Note that it is *not* a requirement of the assignment to develop a Graphical User Interface for this application - where code other than Solidity code is necessary, invocation using command line instructions suffices to meet the requirements.

1. (8 marks) A report (pdf format) describing your design and implementation of the overall system. The report should
    - Describe the data model, and how you have chosen to implement this design using Solidity data structures.
    - In case use of the smart contract requires off-chain computations not implemented in the smart contract, explain what this code does and how to compile and/or operate it. You are not required in this assignment to develop a fancy user interface for any such code: some simple command-line scripts suffice.

- For each of the requirements above, briefly indicate where and how your code meets the requirement. Briefly explain each of the functions in your code. In case you identified any missing requirements or specification ambiguities in the course of your analysis of the application, state what these are and what you have done to resolve and implement them.
- Provide an analysis of the running costs in gas and Australian dollars of the application from the point of view of the buyers and sellers, and how this depends on factors such as the total number of bids and offers. Take into account current information on the costs of transactions on the Ethereum public blockchain, and the gas costs of running your code.
- Describe any security considerations concerning the system and its operation that you consider should be pointed out to the users. Are there any specific traps that the user needs to avoid in using the system, and if so, what are strategies that the user can apply to avoid these traps.
- Explain how your code avoids common Solidity security vulnerabilities like reentrancy attacks.
- Reflectively discuss the overall suitability of the Ethereum platform for this application.

Proper acknowledgement of any sources of information or libraries you have used in your project is required.

2. (10 marks) Submit a directory with all Solidity and ancillary code necessary to build and run your implementation using Truffle.[1] In particular there should be a directory **contracts** containing smart contracts in Solidity for your implementation of the basic functionality. Your code should be well documented.

   If you have used any public Javascript libraries, to avoid an overly large submission file, do not include these, but ensure that there is sufficient information in your submission that these can be automatically installed. In particular, include your `package-lock.json` file.

3. (7 marks) Include a directory **test** containing test cases to validate the correctness of your smart contract implementation. Your report should describe the approach that you have taken to testing, and summarize the test scenarios that you have constructed. It should be possible to execute your tests using the Truffle testing framework. You may also test by running a Ganache instance of the Ethereum blockchain. If specific command line arguments are required to run your tests, include a script

[1] You may choose to do initial development in Remix, but loading your code into Remix creates additional work and inconvenience for the grader, so you should submit in a format that allows testing to be done using Truffle.

that allows the appropriate calls to be made easily by the marker. In any case, the report should contain all the information that is required to determine how to run your tests.

4. (10 marks) Once you have met the basic requirements, attempt the stretch goal. If you attempt this part, your report should contain a section describing what you have done for this part, you should include test cases for this functionality, and it should be clear from your report how to run these test cases.

Resources and Hints for Testing

For Part 1, it will probably be possible to write tests for your code entirely using test scripts written in Solidity. Part 2 is more challenging, since it requires constructing ECDSA signatures, for which Solidity does not provide good support. (The built-in function `ecrecover` only does signature verification.) For this, it is better to write tests in Javascript. In general, JavaScript testing is the more powerful approach (better supported because off-chain application code for interacting with the Ethereum blockchain is most commonly written in JavaScript) and there are a number of JavaScript libraries that support testing. The following resources may be useful when testing your project

- Truffle Documentation `https://trufflesuite.com/docs/truffle/`. See, in particular, the section "Debug and Test".
- The truffle-assertions library `https://www.npmjs.com/package/truffle-assertions`
- Ganache time-traveller `https://www.npmjs.com/package/ganache-time-traveler` helps to test time-based properties.

Your test scenarios should not only test smart contracts with expected input/output, but also check how it handles errors and reverts. For example, you should check that your code does the right thing in situations such as the buyer not having enough funds in their account to pay for a bid that they have made.

Submission

This assignment is required to be your individual work. Submit your project from your CSE account using the command

```
give cs6451 assignment2 <tarfile>
```

on a CSE server.