COMP90088: Cryptocurrencies and decentralised ledgers Semester 1 2022

Project 2 - Auctions in Solidity

due: 2022-05-12, 22:00 AEST

Overview

In this project, you'll develop some Ethereum contracts to implement decentralised auctions. This is a classic application for a smart contract platform and can be implemented (in basic form) in a very small amount of code. You can complete this assignment individually or in pairs.

You'll construct several contracts which gradually increase in complexity and functionality. You should use Solidity, the JavaScript-like language for Ethereum, to implement each part of this project, using the starter code and test code provided. It's important that you implement the interface exactly as specified, as your code will be automatically evaluated. The test code provided gives complete coverage of what you'll be graded on, so if you can pass all tests you should be in line for full credit. Of course, grading will use modified test code with different constants, so if you just hard-code responses this will not work.

Development environments and starter code

The starter code is available at: https://github.com/jcb82/solidityAuctionHouse

We have provided test code and instructions for development using **Truffle** a Node.js-based framework for Solidity/Ethereum development that allows you to run a simulated test network locally. Truffle is generally reliable and has been the most commonly used tool in the burgeoning Ethereum industry.

There are other development environment options which you are welcome to try using, but it will be up to you to port the test code to this environment and debug any problems. We can only provide support for Truffle. Your code will be expected to have the same functionality in all cases.

- **Hardhat** is rapidly gaining popularity as a competitor to Truffle, which supports more graphical features and plugins. This may emerge as the new standard, replacing Truffle.
- **Brownie** is a Python-based competitor which also supports the Vyper language.
- **Remix** is a web-based IDE for Solidity. This is quick to get up and running (no software to install!) and has many features including a GUI debugger. However, the browser IDE is temperamental and it is quite hard to automate tests. Expect your browser to crash somewhat regularly. There is also a feature to save your work in browser storage, but please do not rely on this absolutely and back your work up. Overall this

Part 1: Arbitration

TLDR: Use a designated judge to arbitrate disputes

In class and in the textbook, we discussed fair exchange Bitcoin transactions using a 2-out-of-3 multisig address and a trusted 3rd party judge. The judge can resolve disputes, but in normal cases the buyer and seller can release funds themselves. In Ethereum, there are no multisig addresses so we'll need to achieve the same effect using a smart contract. Our goal is to support arbitration in a seamless way that will work for all three auction types you develop in Parts 2-4. To do this, you'll modify the `Auction.sol` contract, an abstract contract which specific auction formats will inherit from.

Arbitration is *optimistic* in that the judge (if specified) does not need to be involved in the common case. After the auction ends, either the buyer themselves can call `finalize()` to release payment to the seller (if the item is received successfully) or the seller can call `refund()` to return the money to the buyer if they are unable to complete the transaction. If a judge address is specified, the judge also has the ability to call either `refund()` or `finalize()` to adjudicate a dispute on behalf of the buyer or seller, respectively. If no judge is specified, then your contract should default to favoring the seller: the seller can call `refund()` if desired, or anybody can call `finalize()`.

Details:

- Make sure that you identify the caller using `msg.sender` and not `tx.origin`
- You will need to implement the `withdraw()` function, which should be the only place your auction contracts ever send money out. Withdrawal is all-or-nothing, if the contract "owes" a user some money and they call `withdraw()` they should receive all of it.
- Don't allow any calls to call `finalize()` or `refund()` before the auction is over.

Part 2: Dutch auction

TLDR: Price descends until some bidder is willing to pay it

A *Dutch auction*, also called an *open-bid descending-price auction* or *clock auction*, is a type of auction in which the price of the offering (the item for sale) is initially set to a very high value and then gradually lowered. The first bidder to make a bid instantly wins the offering at the current price. There may be a non-zero *reserve price* representing the minimum price the seller is willing to sell for. The offering can never be sold for less than the reserve price, which prevents the auction from being won at a price that is lower than what the offering's owner is willing to accept.

For example, an in-person Dutch auction might start with the auctioneer asking for $1,000. If there are no bidders, the auctioneer might then ask for $900 and continue to lower by $100 every few moments.

Eventually, (say for a price of $500), a bidder will accept and obtain the offering for this last announced price of $500.

For Part 2 you will implement a Dutch auction by filling in `DutchAuction.sol`. This contract will implement the auction of a single offering and will take three parameters at the time of creation (this part is already implemented in the starter code):

- the initial price
- the number of blocks the auction will be open for, including the block in which the auction is created. That is, the auction starts immediately. If the auction starts at time 7 and the length is three, then bids may be submitted at times 7, 8, and 9.
- the (constant) rate at which the price declines per block. Note that the reserve price is implicit given the above three parameters.

Once created, anybody can submit a bid by calling this contract. When a bid is received, the contract calculates the current price by querying `time()` and applying the specified rate of decline in price to the original price.

Important: Do not call `block.number` or `block.timestamp` directly, or otherwise figure out the time using any method but calling `time()`, which is implemented by querying an external contract. This setup allows your code to modularly use a simple notion of time (like `block.number`) or a more complicated version. It also allows the test code to work. This is true for parts 3 and 4 as well.

The first bid which sends a quantity greater than or equal to the current price is the winner. Invalid bids should be rejected (without capturing the bid amount). If a bid is higher than the current price, the excess amount should be returned to the bidder.

Note: Dutch auctions are rarely used in live auction houses, in part because they end abruptly which is less dramatic for the audience. The U.S. Treasury (among others) uses Dutch auctions to sell securities. Dutch auctions have also been used for IPO pricing, most famously by Google. Dutch auctions are theoretically equivalent to sealed-bid first price-auctions. Neither is incentive compatible, as bidders might want to wait to avoid overpaying.

Part 3: English auction

TLDR: New bids increase the price until no new bid has been posted for a fixed number of blocks

Dutch auctions are probably the simplest to implement in a smart contract since only one bid is needed. However, human bidders tend to like auctions with a gradually increasing price as it is more exciting (and may lead impulsive bidders to pay a higher price due to the sense of competition).

For Part 2, implement an *open-bid ascending-price auction*, also called an *English auction*. This is the classic auction house format: the auctioneer starts with an initial offering price (the reserve price) and

asks for an opening bid. Once some individual has placed a bid at or above this price, others are free to offer a higher bid within a short time interval. This is usually where the auctioneer will say “Going once, going twice...” before declaring the offering sold to the last bidder.

You’ll implement a version of this by editing `EnglishAuction.sol`. Like with Part 2, starter code is already provided which captures the key parameters:

- An initial price (the minimum the offering can be had for).
- A bidding period, which is the amount of time bidders have to outbid the current highest bid before the item is gone.
- A minimum increment by which any subsequent bid must increase over previous bids.

Any new bid must be higher than the current highest bid by at least the minimum bid increment. In a live English auction, the auctioneer would probably frown at you if you tried to bid $500.01 after somebody else just bid $500. Without a minimum bid increment in a smart contract, the auction might last a very long time if somebody drove up the price by one wei in each block.

Once created, your contract should accept bids from anybody with a value greater than or equal to the current winning bid plus the minimum bid increment. The initial bid must be the reserve price or higher. When a successful bid is placed, the money should be held by the contract and the value of the previous bid made available immediately for withdrawal by the previous bidder. At the end, when a sufficient number of blocks pass with no new bids, the auction should stop accepting new bids, at which point `getWinner()` should return the highest bidder (requiring you to override the default implementation, which the starter code provides a stub for).

Part 4: Vickrey auction

TLDR: Bidders submit sealed bid commitments and later reveal them. Highest revealed bid wins but pays only the price of the second highest revealed bid. Bidders who don’t reveal forfeit a deposit.

The English auction format has the advantage over the Dutch auction format that the winner doesn’t always need to pay the maximum amount they would have been willing to pay—they just need to outbid the competition. The bidding process enables participants to slowly reveal their willingness to pay. However, this might take a while (particularly if the item is priced too low to start or the bid increment is made too small). By contrast, Dutch auctions are faster, but the winner may end up paying more since they don’t know how much other bidders value the offering and thus may need to be conservative to ensure they win.

In response, bidders in a Dutch auction might not bid in accordance with their true willingness to pay. If they believe nobody else will bid above *x*, they shouldn’t bid too much above *x* lest they overpay. For this reason, Dutch auctions are not *incentive compatible*. Note that early in the course we discussed incentive compatibility in the context of Nakamoto consensus and other distributed protocols, but the concept was initially developed to describe auctions.

Your final task is to implement an auction format which provides the best of both worlds: it can be arbitrarily fast, yet it is incentive compatible: all bidders are incentivized to reveal their true willingness to pay, while the winner only pays as much as they would have in an English auction. This is called a *sealed-bid second-price auction* or *Vickrey auction*. While it's named for economist William Vickrey who first formally described it in 1961 and helped popularize it, this format has been used in practice since at least the 19th century.

An offline Vickrey auction proceeds as follows: all participants submit their bid in a sealed envelope. The auctioneer then opens all of the envelopes, and the highest bidder obtains the offering but only pays the price specified by the second-highest bidder (hence the term second-price auction). You can see intuitively why this is equivalent to the outcome an English auction would have eventually produced: this is the price the highest bidder would have needed to pay (perhaps plus a small increment) to outbid their closest competitor in an English auction.

You will implement this by editing `VickreyAuction.sol`. The contract takes as parameters:

- A minimum price. Bids below this are ignored (but the deposit is returned).
- A time period to submit bid commitments.
- A time period to reveal bids.
- A bid deposit amount.

The auction will have two phases:

1. During the bid submission phase (which lasts until `biddingDeadline`, exclusive), anyone can submit a bid. To preserve secrecy, bidders submit a commitment to their bid, rather than the bid itself. Specifically, this commitment should be a SHA3[1] hash of a 32-byte nonce and their bid value (formatted as a 32-byte array). You should make sure your code accepts the commitments as generated by the provided test code, which uses `abi.encodePacked()` to combine these two inputs into one argument to the hash function:

   ```
   bytes32 commitment = keccak256(abi.encodePacked(_bidValue, nonce));
   ```

   Bidders must also send the value specified by the bidding deposit requirement. This money is to ensure they submit a well-formed bid and eventually open their bid. Bidders must provide the deposit amount exactly with their first bid, any other amount should be rejected. Bidders may update their bid for free. Bidders will get their deposit back in the second phase after revealing their bid.
2. In the bid opening phase (which lasts from the `biddingDeadline` inclusively until the `revealDeadline` exclusively) all bidders should reveal their bid by sending the nonce used in their bid commitment. They must also send the precise amount of funds to the contract to pay for their bid if they win. Any attempts to open a bid incorrectly (e.g. by sending an incorrect nonce or failing to send the correct funding value) must be rejected. Every valid bid opening

[1] Solidity refers to the SHA3 hash function by its older name, Keccak256.

should receive the bid deposit back. *Security warning:* ensure that only the original bidder can receive their bid deposit back and they can only do so once.

Finally, after the bid opening phase has ended, the winner is whoever sent in the highest bid. The winning price, of course, is the second highest bid value which was revealed (or the reserve price if only one valid bid was opened). After the reveal deadline has passed, the `finalize()` function will need to be called as with the English auction. Of course, if called too early it should not close the auction. The same mechanics for the judge apply, if a judge is specified.

All losing bidders (who successfully opened their bids) must be able to withdraw their bid values (in addition to their bid deposits). The winner also likely also needs to withdraw a partial refund, since they sent in the full amount of their bid but only pay the amount of the second-highest bid. This should happen when `finalize()` is called. Make sure that if there is only one bidder who opens a committed bid successfully, they are refunded the difference between their bid and the minimum price.

Note that in practice, the bid deposit should be quite high (on the order of the expected price of the offering). If it is too low, the winning bidder might try to bribe the losers not to reveal their bids, lowering the price they ultimately pay. The bid deposit should be high enough that losers are always strongly incentivized to reveal their bids.

Deliverables

You only need to submit only the contract files, **`Auctions.sol, DutchAuction.sol, EnglishAuction.sol`** and **`VickreyAuction.sol`** and a **`README`** containing the student names and student numbers of the group members. Remember, your code will be automatically graded so it is imperative that you don't change the API of existing functions (adding functions won't hurt).

Truffle instructions

We have provided testing code using the Truffle development environment. Here are the instructions to install the required dependencies and run the tests:

1. Install Node.js (if you haven't already)
2. Install the following dependencies for this project
   `npm -g install truffle`
3. Clone or download the starter code (do not clone into a public repository!)
4. Create a new Truffle project (enter N thrice to not overwrite contracts, migrations or test)
   `truffle init`
5. From the starter code, you can find the appropriate files and put them into your folder.
   a. The `contracts` folder contains Solidity files ending in .sol. These will be the main files you edit.
   b. The `test` folder contains testing files, also written in Solidity. You may modify these for enhanced testing.
   c. The `migrations` folder contains more Javascript, which you shouldn't need to edit.
6. To compile your code and run the provided test files. To run an individual test:
   `truffle test test/ArbitrationTest.sol`
   To run all tests that Truffle can find:
   `truffle test`
   To compile only:
   `truffle compile`

Tips:

- If your code throws an exception where the test code is not expecting it, you may get quite unhelpful feedback.
- There is a debugger, and a command line interface to get the state of a transaction. Your mileage may vary on how useful these are.