CS 251: Bitcoin and Cryptocurrencies Fall 2016

Project 4

due: 2016-12-07, 23:59 via email to cs251ta@cs.stanford.edu

Overview

In this project, we will develop an auction system using Ethereum contracts. This is one example of a trustless decentralized escrow system, in which smart contracts provide logic that regulates transfers between parties. This is a powerful use case for smart contracts in general and Ethereum provides a good platform to implement this.

We will 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 available at https://crypto.stanford.edu/cs251/proj/proj24.tar.gz. Your code will be auto-graded, so it is important that you maintain the exact API in the starter code.[1]

Part 1: 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 (good) 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 1 you will implement a Dutch auction using Ethereum. This contract will implement the auction of a single offering and will take four parameters at the time of creation:

- the reserve price (which may be zero)
- the address of a judge for payment finalization (to be ignored until part 2)
- 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.

[1]If you really like living on the edge, you can develop in another programming language that compiles to EVM, but you'll need to ensure compatibility with the ABI of the Solidity starter code provided.

- the (constant) rate at which the price declines per block. Note that the initial price is derived from the reserve price, the decrement and the length of the bidding period.

Once created, anybody can submit a bid by calling this contract. When a bid is received, the contract calculates the current price by querying the current block number and applying the specified rate of decline in price to the original price. The first bid which sends a quantity greater than or equal to the current price is the winner; the money should be immediately transferred to the seller (the party which created the contract) and the auction contract terminated. Invalid bids should be refunded immediately.

Note: Dutch auctions are rarely used in live auction houses, perhaps 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 as an alternative bidding process for IPO pricing, most famously by Google. Dutch auctions are theoretically equivalent to sealed-bid first price-auctions. They are not incentive compatible, as bidders might want to wait to avoid over-paying.

Part 2: Adding a judge

TLDR: Designate a third-party judge and prevent payment to the seller until the winner approves. If needed, the judge can force the winner to pay or refund them.

In class and in the textbook, we saw 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 add support for a judge in a seamless way that your Dutch auction implementation can use, as well as those you will implement in Part 3 and Part 4. Modify the implementation of `finalize()` in the DAuction contract so that if a judge address is specified (in the parameter `_judgeAddress`), the contract will require an explicit call to `finalize()`. In the normal case, this can be called by the winning bidder to acknowledge receipt of the goods. Note that in the case of a Dutch auction, you still want to terminate immediately if no judge is specified.

However, If the winning bidder does not call `finalize()`, the judge is able to step in and call this function to forcibly transfer all funds to the seller. Alternately, the judge (and the judge alone) can call the `refund()` function, passing in an amount specifying a partial or full refund back to the winning bidder. After either a call to `finalize()` by the judge or winning bidder or a call to `refund()` by the judge, the contract should terminate. Make sure that the judge can only call one of these functions once and make sure the judge cannot send the money to themselves!

Your judge logic will be used for Part 3 and Part 4 as well.

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 just 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 usually where the auctioneer will say "Going once, going twice..." before declaring the offering sold to the last bidder.

Implement a contract that supports a version of this. Like with Part 1, the contract should only implement the auction of a single offering. In addition to the reserve price and judge address, interpreted exactly as before, your English auction contract will take 2 additional parameters:

- the number of blocks a bid must be unchallenged before it is considered the winner, including the block in which the bid is posted.
- the minimum bid increment

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 forever as somebody drives 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 returned to the previous bidder. At the end, when a sufficient number of blocks pass with no new bids, the auction should stop accepting new bids and wait for the `finalize()` function to be called. If there is no judge, anybody should be able to call this function. In practice either the buyer or seller might call it to get the transaction moving. Like before, if a judge is specified then only the judge or buyer can finalize.

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 auction 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. We're truly coming full circle!

In this section we'll 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.

Your Ethereum implementation of a Vickrey auction will take similar parameters as before: a reserve price and an optional judge's address. It will also take a bid commitment phase length, a bid opening phase length and a bid deposit requirement. The auction will have two phases:

1. During <u>the bid submission phase</u> (which lasts commitTimePeriod blocks, including the block the auction was created) 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 hash of a 32 byte nonce and their bid value (formatted as a 256-bit buffer). You should make sure your code accepts the commitments generated by the provided test code. Hint: in Solidity, you can call `sha3()` with an arbitrary number of inputs and their values will simply be concatenated and hashed. 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 will get their deposit back in the second phase after revealing their bid.
2. In <u>the bid opening phase</u> (which lasts revealTimePeriod blocks, starting from the block after the last block in which bids can be submitted) 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 should receive the bid deposit back. *Security warning:* make sure that only the original bidder can receive their bidding deposit back and they can only do so once.

Finally, after the bid opening period has ended (at the specified time), the winner is the entity which 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) need to get their bid amounts refunded (in addition to their bid deposits). You may do this all in one pass at the time the auction closes, or you may optimize by doing this during the bid opening phase. If a higher bid has already been opened, losing bidders can be immediately refunded (or need not send in their actual bid amount at all). It's up to you how you implement this, but losing bidders must be refunded.

The winner also likely also needs a 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.

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. The contract may be left with surplus funds at the end due to unopened bids. You may burn this money, or send it to a designated charity address if you wish. You should *not* send it to the winning bidder: otherwise, they can effectively use it to bribe other bidders not to open their bids.

Testing & Security

We've provided partial test code for parts 1, 3, and 4. You'll probably want to modify this code and add more tests-passing these tests is necessary but not sufficient for a correct implementaiton.

You are also responsible for writing secure code, which the partial tests do not cover. Your score may be reduced if your solution is vulnerable to known classes of bugs such as stack exhaustion attacks (a type of unchecked send bug) or reentrancy attacks. A very detailed security guide is available here.

Deliverables

You only need to submit only the contract files, **`DAuctions.sol, Dauction.sol, DDutchAuction.sol, DEnglishAuction.sol, DVickreyAuction.sol`**, by sending them to cs251ta@cs.stanford.edu as usual. Remember, this function will be automatically graded so it is imperative that you don't change the API of existing functions (adding functions won't hurt). Make sure to follow the naming format for the submission folder as per previous projects.

Environment and testing

You may develop and test your code however you like. For example can use the in-browser Solidity compiler, but it has quite limited testing facilities and you're unlikely to produce a quality solution without additional testing.

We have provided a testing framework in the starter code using the Truffle development environment. Below you have been provided the links with instructions to install each required package, although different machine configurations might require different steps.

Here are the instructions to install the required dependencies and run the tests, assuming you have already installed :

1. Install Node.js (if you haven't already)
2. Install the following dependencies for this project:
    a. Truffle
    b. Mocha
    c. web3
    d. Ethereum RPC
    e. EthereumJS ABI
    f. EthereumJS Utils
    g. Node.js crypto

    To do it all in one line:
    ```
        sudo npm -g install truffle mocha web3 ethereumjs-testrpc
    ethereumjs-abi ethereumjs-util crypto
    ```
3. Create a new Truffle project
    ```
        truffle init
    ```
4. From the starter code, you can find the appropriate files and put them into your folder.
    a. Paste all the contract files into contracts folder. (files end with .sol, from contracts folder in the starter code)
    b. Paste all of the test files in the test folder. (files starting with test, from test folder in the starter code)
    c. Paste the deploy file in the migrations folder. (file starts with a number, in the migrations folder in the starter code)
5. Compile the contracts
    ```
        truffle compile
    ```
6. Run testrpc
7. Run migrations using: truffle migrate
8. Run the tests using:
    a. truffle test test/test_english_auction.js
    b. truffle test test/test_dutch_auction.js
    c. truffle test test/test_vickrey_auction.js