

EN.601.441/641: Assignment 2

October 2024

This is a pair assignment. Submit to Gradescope by October 16th at 11:59pm.

Overview

In this project, you'll develop some Ethereum contracts to implement decentralized 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'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 environment and starter code. For this assignment you should use Remix, the web-based IDE for Solidity. The starter code for the assignment is available [here](#).

The link above will load the test framework, and the template files, into the IDE. In order to work on the assignment and compile the tests, you first need to create the following files in the top level of the file browser: Auction.sol, DutchAuction.sol, EnglishAuction.sol, and VickreyAuction.sol. The file explorer should look like the image below. Copy-paste the starter code from the corresponding files, such as "Auction-template.sol".

Finally, you need to change the second line of Auction.sol. Change the line from `import gist/Timer.sol;` to `import "gist-6c144b00ae9e26fbbb93dc0fd5acfa89/Timer.sol";`.

Local Development. Unfortunately, Hopkins blocks Ethereum-related websites on the campus network, preventing access to the online Remix IDE. If you would like to work on this assignment while on the Hopkins campus, you should download the local version of the IDE.

Then, you can download the starter code directly from the [gist](#).

With the downloaded code, you should set up your development environment as shown in the screenshot above, and continue to develop locally.

Part 1: Dutch auction

<i>TLDR: the price descends until some bidder is willing to pay it.</i>

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. After a set time, the auction may end without a winner if the price never goes low enough to attract a bid.

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 five parameters at the time of creation:

- the address of the “seller” of the auction
- the address of a Timer contract, used by the contract to check time
- 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.
- the (constant) rate at which the price decrements per block. Remember, in a Dutch auction the price gets cheaper the longer the auction is open for.

Once created, any address can submit a bid by calling the `bid()` function for 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. Any invalid bids should be refunded immediately. The auction’s creator can call `finalize()` after completion to destroy the contract and collect its funds.

Historical 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.

Etheruem note: Dutch auctions are theoretically equivalent to sealed-bid first price-auctions; bidders should bid as soon as the price reaches their maximum willingness to pay to avoid being scooped. In practice, they are not incentive compatible, as bidders might want to wait to avoid over-paying. Think carefully about the implications of running a Dutch auction in Ethereum. Would buyers really need to offer a price based on their true willingness to pay if they suspect nobody else values the item as much?

Part 2: 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.

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 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 minimum 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, 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 3: 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 is probably the one you know the best: 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.

This 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. It is incentive compatible (in the absence of bidder collusion): there is no need for bidders to reason about their competitor's willingness to pay as the bidding process enables participants to slowly reveal their willingness to pay (price discovery).

However, this auction format can 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 not incentive-compatible.

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 (famously, by stamp collectors).

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.

Your Ethereum implementation of a Vickrey auction will take the following parameters:

- a reserve price, the minimum the offering can be obtained for. Bids below this are rejected.
- the number of blocks the bid submission phase will be open for, including the block in which the auction is created. That is, the auction starts immediately.
- the number of blocks the bid opening phase will be open for, starting immediately after the bid submission phase ends.
- the deposit amount required to submit a bid. This is necessary from the auctioneer's perspective to ensure that losing bidders have a strong incentive to open their bid and discover the true second price. Otherwise a winning bidder might bribe losers not to reveal their bid in order to get the item at a lower price.

The auction will have two phases: During the bid submission phase (which lasts `biddingPeriod` 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 use the provided `makeCommitment()` function to make or verify bid commitments.

```
function makeCommitment(uint bidValue, bytes32 nonce) public pure returns(bytes32) {
    return keccak256(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 will get their deposit back in the second phase after revealing their bid. In the bid opening phase (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 Dutch auction. Of course, if called too early it should not close the auction.

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.

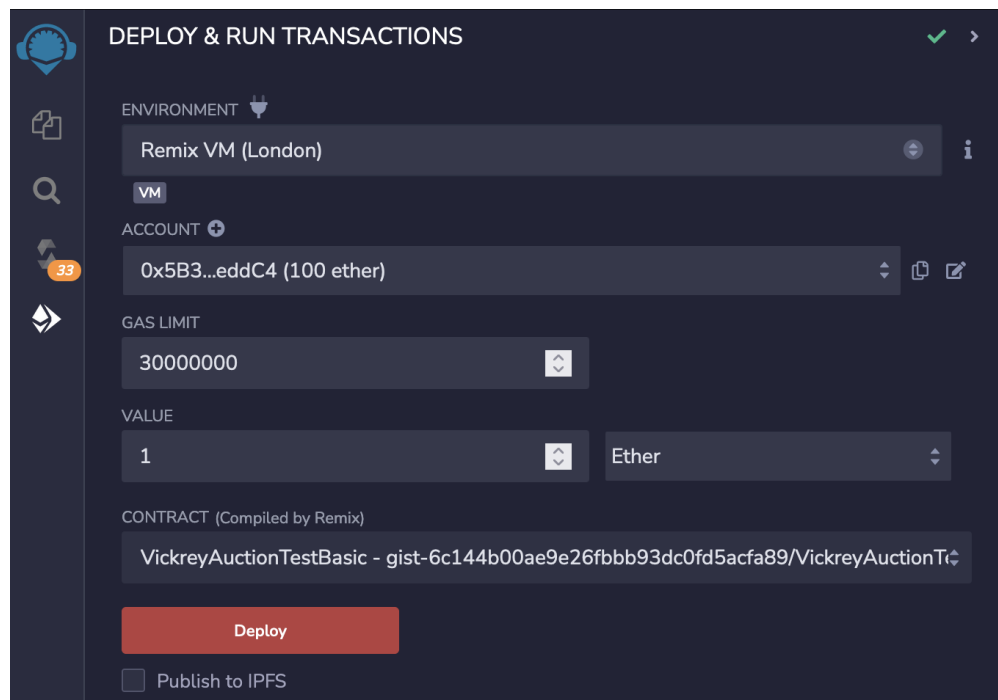
Deliverables

You only need to submit only the four contract files, `Auctions.sol`, `DutchAuction.sol`, `EnglishAuction.sol`, and `VickreyAuction.sol` via Gradescope. 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).





How to run tests in Remix

Remix should work out-of-the-box in a modern browser. However, running your code is somewhat trickier. There is no simple way to run all of the tests (due to gas limits and other artifacts).




To run a test, you'll need to compile the contract that contains the test you want to run (e.g. `VickreyAuctionTestBasic.sol`). You can do so by right-clicking the file in the file browser and selecting "compile." Then, click the Ethereum symbol in the left nav bar to go to the "deploy and run transactions" browser. You will also need to make sure to give the contract an initial balance of at least 0.001 ether. For `VickreyAuctionTestAdvanced.sol`, you will need to manually raise the gas limit for this to work. The parameters in the following image should work for each test contract.



After deploying your contract (by pressing the deploy button) you'll want to run the tests. To test your code, you'll need to run each individual test case by clicking the name of the function, like "testCommitBids", to call the test function. If this runs without throwing an error (showing a green checkmark), it means your code is passing the test. Otherwise, you will need to use the debugger to interpret the error.



33



DEPLOY & RUN TRANSACTIONS

Deployed Contracts

VICKREYAUCTIONTESTBASIC AT 0XD91...39138 (MEMORY)

Balance: 1 ETH

Assert_equal

bytes32 value1, bytes32 value2, string message

Assert_equal

uint256 value1, uint256 value2, string message

Assert_equal

address value1, address value2, string message

Assert_isFalse

bool value, string message

Assert_isTrue

bool value, string message

setupContracts

testChangeBid

testCommitBid

testCreateVick

testEarlyRevea

testExcessBidC

✓

[vm] from: 0x583...eddC4 to: VickreyAuctionTestAdvanced.setupContracts() 0xd91...39138 value: 0 wei
data: 0x215...a9925 logs: 0 hash: 0x53c...ab3fa
transact to VickreyAuctionTestAdvanced.testMinimalBidder pending ...

transact to VickreyAuctionTestAdvanced.testMinimalBidder errored: VM error: revert.

revert

The transaction has been reverted to the initial state.
Note: The called function should be payable if you send value and the value you send should be less than your current balance.
Debug the transaction to get more information.

✗

[vm] from: 0x583...eddC4 to: VickreyAuctionTestAdvanced.testMinimalBidder() 0xd91...39138 value: 0 wei
data: 0x957...483f1 logs: 0 hash: 0xcdb...fcd92

Debug

▼

6