1 VickreyAuction

```
pragma solidity ^0.4.22;
import "./SimpleEscrow.sol";


contract VickreyAuction{
  // length of each of the 3 phases expressed in mined blocks
  uint256 commitmentPhaseLength;
  uint256 withdrawalPhaseLength;
  uint256 openingPahseLength;
  // finalize function can be called only one time
  bool finalizeCalled = false;
  uint256 reservePrice;
  uint256 depositRequired;
  address seller;
  mapping(address => uint256) commitedEnvelops;

  // used for escrow
  address escrowTrustedThirdParty;
  SimpleEscrow simpleescrow;

  uint256 gracePeriod;

  uint256 firstBid;
  address firstBidAddress;
  uint256 secondBid;
  address secondBidAddress;

  // events
  event AuctionCreated(uint32 availableIn); // getting the number of blocks corresponding to the grace
      period
  event CommitedEnvelop(address bidderAddress);
  event Withdraw(address leavingBidderAddress);
  event Open(address bidderAddress, uint256 value);
  event FirstBid(address bidderAddress, uint256 value);
  event SecondBid(address bidderAddress, uint256 value);
  event Winner(address winnerBidder, uint256 value);

  // testing related event
  event NewBlock(uint256 blockNum);


  constructor (uint256  _reservePrice,
        uint256 _commitmentPhaseLength,
        uint256 _withdrawalPhaseLength,
        uint256 _openingPahseLength,
        uint256 _depositRequired,
        address _seller,
        uint32 miningRate) public{
    require(_seller != msg.sender, "the seller can't create the auction");
    // the deposit must be at least two times the reservePrice
    require(_depositRequired >= 2*_reservePrice,"deposit must be >= 2*reservePrice");

    seller = _seller;
    reservePrice = _reservePrice;
    commitmentPhaseLength = _commitmentPhaseLength;
    withdrawalPhaseLength = _withdrawalPhaseLength;
    openingPahseLength = _openingPahseLength;

    depositRequired = _depositRequired;
    // the auction house will also be the trusted third party for the escrow
    escrowTrustedThirdParty = msg.sender;

    // miningRate == 15 means that on average one block is mined every 15 seconds
    gracePeriod = block.number + 5*60 / miningRate;

    /**
    * Since the first bid is init to reservePrice the bidder to win has to al least offer reservePrice + 1
    * The semantic is not perfect, but keeping it in this way allows the code to be more clean
    */
    secondBid = firstBid = reservePrice;
    secondBidAddress = firstBidAddress = this;

    emit AuctionCreated( 5*60 / miningRate);
```

```
}


modifier checkCommitmentPahseLenght(){
  require(gracePeriod < block.number &&
  block.number <= gracePeriod + commitmentPhaseLength, "not in commitment phase");_;
}
modifier checkWithdrawalPhaseLength(){
  require(gracePeriod + commitmentPhaseLength < block.number &&
  block.number <= gracePeriod + commitmentPhaseLength + withdrawalPhaseLength, "not in withdrawal phase"
      );_;
}
modifier checkOpeningPahseLength(){
  require(gracePeriod + commitmentPhaseLength + withdrawalPhaseLength < block.number &&
  block.number <= gracePeriod + commitmentPhaseLength + withdrawalPhaseLength + openingPahseLength, "not
      in opening phase");_;
}
modifier checkAuctionEnd(){
  require(gracePeriod + commitmentPhaseLength + withdrawalPhaseLength + openingPahseLength < block.
      number, "auction not ended");_;
}


function commitBid( uint256 envelop) external payable checkCommitmentPahseLenght(){
  require(msg.sender != escrowTrustedThirdParty,"escrow third party can't commit bid");
  // the seller can't bid
  require(msg.sender != seller, "seller can't commit bid");
  // won't keeep the actual deposit given,  the sender should send the right ammount
  require(msg.value >= depositRequired, "need to send the deposit" );
  // the bidder can't bid more than one time
  require(commitedEnvelops[msg.sender] == 0, "you already called this function");

  commitedEnvelops[msg.sender] = envelop;
  emit CommitedEnvelop(msg.sender);
}


function withdraw() external checkWithdrawalPhaseLength(){

  if (commitedEnvelops[msg.sender] != 0){
    // this avoids to call the function multiple times
    commitedEnvelops[msg.sender] = 0;
    msg.sender.transfer(depositRequired/2);
    emit Withdraw(msg.sender);
  }
}


function open(uint256 nonce) external payable checkOpeningPahseLength(){
  // checking if the bid and the evelop match
  uint256 tmphash = uint256(keccak256(msg.value, nonce));
  // the last condition is useful to a avoid a fake withdraw: a bidder can bid 0 then open to get the
      deposit back
  require((commitedEnvelops[msg.sender] == tmphash) && (msg.value >= reservePrice), "haven't sent the
      right ammount" );

  // to avoid more calls of the function from the same bidder
  commitedEnvelops[msg.sender] = 0;
  emit Open(msg.sender, msg.value);

  uint256 oldSecondBid;
  address oldSecondBidAddress;
  if(msg.value > firstBid){
    oldSecondBid = secondBid;
    oldSecondBidAddress = secondBidAddress;

    // first bid will become the secondBid, the old second bidder will be refounded
    secondBid = firstBid;
    secondBidAddress = firstBidAddress;
    emit SecondBid(secondBidAddress, secondBid);

    firstBid = msg.value;
    firstBidAddress = msg.sender;
    emit FirstBid(msg.sender, msg.value);

```

```
      // prefer first to set the values and then to transfer the money
      if (oldSecondBid > reservePrice)
      oldSecondBidAddress.transfer(oldSecondBid + depositRequired);

    }else if (msg.value > secondBid) {
      oldSecondBid = secondBid;
      oldSecondBidAddress = secondBidAddress;

      secondBid = msg.value;
      secondBidAddress = msg.sender;
      emit SecondBid(msg.sender, msg.value);

      if (oldSecondBid > reservePrice)
      oldSecondBidAddress.transfer(oldSecondBid + depositRequired);

    }else
    msg.sender.transfer(msg.value + depositRequired);
  }


  function finalize() public checkAuctionEnd(){
    require(finalizeCalled == false, "finalize function already called");
    // only the auction house can call this
    require(msg.sender == escrowTrustedThirdParty, "only the auction house can call this function");
    finalizeCalled = true;

    if(firstBid == reservePrice) // noone has betted
    return;

    // TIE RESOLUTION RULE if there is a tie the first one who opened the envelop wins
    if (secondBid != reservePrice)
    secondBidAddress.transfer(secondBid + depositRequired);

    // the winner pays the ammount offered by the second winner
    emit Winner(firstBidAddress, firstBid);
    firstBidAddress.transfer(firstBid - secondBid + depositRequired);

    //burning remaining ether
    address burnAddress = 0x0;
    burnAddress.transfer(address(this).balance - secondBid);

    //seller.transfer(secondBid);
    simpleescrow = new SimpleEscrow(seller,firstBidAddress,escrowTrustedThirdParty);
    address(simpleescrow).transfer(secondBid);
  }

  // escrow related wrappers
  modifier checkEscrowSender(){
    require(msg.sender == seller || msg.sender == firstBidAddress || msg.sender == escrowTrustedThirdParty
        );_;
  }

  function acceptEscrow() public checkAuctionEnd() checkEscrowSender(){
    require(finalizeCalled == true);

    simpleescrow.accept(msg.sender);
  }

  function refusetEscrow() public checkAuctionEnd() checkEscrowSender(){
    require(finalizeCalled == true);

    simpleescrow.refuse(msg.sender);
  }

  function concludeEscrow() public checkAuctionEnd(){
    require(finalizeCalled == true);
    // only the trusted third party can conclude
    require(msg.sender == escrowTrustedThirdParty);

    simpleescrow.conclude();
  }

  // getters
  function getReservePrice() public view returns(uint256){
    return reservePrice;
  }
```

```
  function getSeller() public view returns(address){
    return seller;
  }

  function getDepositRequired() public view returns(uint256){
    return depositRequired;
  }

  function getTrustedThirdParty() public view returns(address){
    return escrowTrustedThirdParty;
  }

  // functions to know how many blocks are left to the end of each phase
  function getGracePeriod() public view returns(uint256){
    require(block.number <= gracePeriod);
    return gracePeriod - block.number;
  }

  function getCommitmentPhaseLength() public view  checkCommitmentPahseLenght() returns(uint256){
    return gracePeriod + commitmentPhaseLength - block.number;
  }

  function getWithdrawalPhaseLength() public view  checkWithdrawalPhaseLength() returns(uint256){
    return gracePeriod + commitmentPhaseLength + withdrawalPhaseLength - block.number;
  }

  function getOpeningPhaseLength() public view checkOpeningPahseLength() returns(uint256){
    return gracePeriod + commitmentPhaseLength + withdrawalPhaseLength + openingPahseLength - block.number
        ;
  }

  /*
  * The function above are added only to test better the contract.
  * In a real environment they should be removed
  */
  function addBlock() public{
    emit NewBlock(block.number);
  }

  /*
  * Can be used to retrive the hash to be passed to the open function
  */
  function doKeccak(uint256 value, uint256 nonce) external pure returns(uint256) {
    return uint256(keccak256(value,nonce));
  }

}
```

2 SimpleEscrow

```
pragma solidity ^0.4.22;

/**
* This contract and it's code must be called inside a Auction contract
*/
contract SimpleEscrow{
  address winnerBidder;
  address seller;
  address trustedThirdParty;
  address auctionContract;

  bool winnerBidderAccepted = false;
  bool sellerAccepted = false;
  bool thirdPartyAccepted = false;

  bool winnerBidderRefused = false;
  bool sellerRefused = false;

  event EscrowAccepted(address subj);
  event EscrowRefused(address subj);
  event EscrowClosed();

  constructor (address _seller, address _bidder, address _trustedThirdParty) public payable {
    require(_seller != _bidder && _seller != _trustedThirdParty && _bidder != _trustedThirdParty);
    require(isContract(msg.sender));
    seller = _seller;
    winnerBidder = _bidder;
    trustedThirdParty = _trustedThirdParty;

    auctionContract = msg.sender;
  }
  modifier checkBalance(){
    require(address(this).balance > 0,"first you should send money");_;
  }


  modifier checkSender(){
    // only the creator can call the functions
    require(msg.sender == auctionContract);_;
  }

  function () public payable checkSender() checkBalance(){
    // only the creator can send the contended moneys
  }

  function accept(address addr) public checkSender() checkBalance() {
    if (addr == seller) sellerAccepted = true;
    else if (addr == winnerBidder) winnerBidderAccepted = true;
    else if (addr == trustedThirdParty) thirdPartyAccepted = true;
    emit EscrowAccepted(addr);
  }

  function refuse(address addr) public checkSender() {
    if (addr == seller) sellerRefused = true;
    else if (addr == winnerBidder) winnerBidderRefused = true;
    emit EscrowRefused(addr);

  }

  function conclude() public checkSender() checkBalance(){
    // gonna list all the possible cases
    if (sellerAccepted && winnerBidderAccepted)
      seller.transfer(address(this).balance);
    else if (sellerAccepted && thirdPartyAccepted)
      seller.transfer(address(this).balance);
    else if(winnerBidderAccepted && thirdPartyAccepted)
      winnerBidder.transfer(address(this).balance);

    else if (sellerRefused && winnerBidderRefused)
      winnerBidder.transfer(address(this).balance);

    emit EscrowClosed();
  }
```

```
74
75    // checking if the address is a contract address
76    function isContract(address _addr) private view returns (bool){
77      uint32 size;
78      assembly{
79      size := extcodesize(_addr)
80      }
81      return (size > 0);
82    }
83
84    // getters
85    function getWinnerBidder() public view returns(address){
86      return winnerBidder;
87    }
88
89    function getSeller() public view returns (address){
90      return seller;
91    }
92
93    function getTrustedThirdParty()public view returns (address){
94      return trustedThirdParty;
95    }
96  }
```

3 DutchAuction

```
pragma solidity ^0.4.22;
import "./SimpleEscrow.sol";
import "./DecreasingStrategies.sol";


contract DutchAuction{
  uint256 openedForLength;
  uint256 initialPrice;
  uint256 reservePrice;
  address seller;

  IDecreasingStrategy decrStratedy;

  // used for escrow
  address firstBidAddress;
  address escrowTrustedThirdParty;
  SimpleEscrow simpleescrow;

  uint256 gracePeriod;
  bool bidSubmitted = false;
  // events
  event AuctionCreated(uint32 availableIn); // getting the number of blocks corresponding to the grace period
  event NotEnoughMoney(address bidder, uint256 sent, uint256 price);
  event Winner(address winnerBidder, uint256 bid);

  // testing related event
  event NewBlock(uint256 blockNum);


  constructor (uint256  _reservePrice,
        uint256 _initailPrice,
        uint256 _openedForLength,
        address _seller,
        IDecreasingStrategy _decrStratedy,
        uint32 miningRate) public{
    require(_seller != msg.sender);
    require(_initailPrice > _reservePrice && _reservePrice >= 0);

    openedForLength = _openedForLength;
    seller = _seller;
    initialPrice = _initailPrice;
    reservePrice = _reservePrice;

    decrStratedy = _decrStratedy;

    // the auction house will also be the trusted third party for the escrow
    escrowTrustedThirdParty = msg.sender;

    // miningRate == 15 means that on average one block is mined every 15 seconds
    gracePeriod = block.number + 5*60 / miningRate;


    emit AuctionCreated( 5*60 / miningRate);

  }

  modifier checkPeriod(){
    require(gracePeriod < block.number && block.number <= gracePeriod + openedForLength, "not int the bid phase");_;
  }



  function bid() public payable checkPeriod(){
    require(msg.sender != seller && msg.sender != escrowTrustedThirdParty);
    require(bidSubmitted == false, "someone else has already bidded");
    uint256 currentPrice = decrStratedy.getCurrentPrice(block.number - gracePeriod, openedForLength, initialPrice, reservePrice);

    if(msg.value >= currentPrice){
      bidSubmitted = true;
      firstBidAddress = msg.sender;
```

```
71        emit Winner(msg.sender, msg.value);
72        simpleescrow = new SimpleEscrow(seller,firstBidAddress,escrowTrustedThirdParty);
73        address(simpleescrow).transfer(msg.value);
74      } else {
75        // sending the money back
76        emit NotEnoughMoney(msg.sender,msg.value, currentPrice);
77        msg.sender.transfer(msg.value);
78      }
79    }
80
81    // escrow related wrappers
82    modifier checkEscrowSender(){
83      require(msg.sender == seller || msg.sender == firstBidAddress || msg.sender == escrowTrustedThirdParty
          );_;
84    }
85
86    function acceptEscrow() public checkEscrowSender(){
87      require(bidSubmitted == true);
88
89      simpleescrow.accept(msg.sender);
90    }
91
92    function refusetEscrow() public  checkEscrowSender(){
93      require(bidSubmitted == true);
94
95      simpleescrow.refuse(msg.sender);
96    }
97
98    function concludeEscrow() public{
99      require(bidSubmitted == true);
100     // only the trusted third party can conclude
101     require(msg.sender == escrowTrustedThirdParty);
102
103     simpleescrow.conclude();
104   }
105
106   // getter
107   function getSeller() public view returns(address){
108     return seller;
109   }
110
111   function getReservePrice() public view returns (uint256){
112     return reservePrice;
113   }
114
115   function getInitialPrice() public view returns(uint256){
116     return initialPrice;
117   }
118
119   function getCurrentPrice() public view checkPeriod() returns(uint256){
120     return decrStratedy.getCurrentPrice(block.number- gracePeriod, openedForLength, initialPrice,
          reservePrice );
121   }
122
123   // getting the remaning number of block the auction will be opened for
124   function getOpenedFor() public view checkPeriod returns(uint256){
125     return gracePeriod + openedForLength - block.number;
126   }
127
128   // test purposes only
129   function addBlock() public {
130     emit NewBlock(block.number);
131   }
132
133 }
```

4 Decreasing Strategies

```
pragma solidity ^0.4.22;

contract IDecreasingStrategy{

  function getCurrentPrice(uint256 elapsedTime,
               uint256 totalTime,
               uint256 initailPrice,
               uint256 reservePrice) public pure returns(uint256);

  // logarithm
  // https://ethereum.stackexchange.com/questions/8086/logarithm-math-operation-in-solidity
  function log2(uint x)public pure returns (uint y){
    assembly {
      let arg := x
      x := sub(x,1)
      x := or(x, div(x, 0x02))
      x := or(x, div(x, 0x04))
      x := or(x, div(x, 0x10))
      x := or(x, div(x, 0x100))
      x := or(x, div(x, 0x10000))
      x := or(x, div(x, 0x100000000))
      x := or(x, div(x, 0x10000000000000000))
      x := or(x, div(x, 0x100000000000000000000000000000000))
      x := add(x, 1)
      let m := mload(0x40)
      mstore(m,           0xf8f9cbfae6cc78fbefe7cdc3a1793dfcf4f0e8bbd8cec470b6a28a7a5a3e1efd)
      mstore(add(m,0x20), 0xf5ecf1b3e9debc68e1d9cfabc5997135bfb7a7a3938b7b606b5b4b3f2f1f0ffe)
      mstore(add(m,0x40), 0xf6e4ed9ff2d6b458eadcdf97bd91692de2d4da8fd2d0ac50c6ae9a8272523616)
      mstore(add(m,0x60), 0xc8c0b887b0a8a4489c948c7f847c6125746c645c544c444038302820181008ff)
      mstore(add(m,0x80), 0xf7cae577eec2a03cf3bad76fb589591debb2dd67e0aa9834bea6925f6a4a2e0e)
      mstore(add(m,0xa0), 0xe39ed557db96902cd38ed14fad815115c786af479b7e83247363534337271707)
      mstore(add(m,0xc0), 0xc976c13bb96e881cb166a933a55e490d9d56952b8d4e801485467d2362422606)
      mstore(add(m,0xe0), 0x753a6d1b65325d0c552a4d1345224105391a310b29122104190a110309020100)
      mstore(0x40, add(m, 0x100))
      let magic := 0x818283848586878898a8b8c8d8e8f929395969799a9b9d9e9faaeb6bedeeff
      let shift := 0x100000000000000000000000000000000000000000000000000000000000000
      let a := div(mul(x, magic), shift)
      y := div(mload(add(m,sub(255,a))), shift)
      y := add(y, mul(256, gt(arg, 0x8000000000000000000000000000000000000000000000000000000000000000)))
    }
  }

}


contract LinearDecreasingStrategy is IDecreasingStrategy {

  function getCurrentPrice(uint256 elapsedTime,
               uint256 totalTime,
               uint256 initailPrice,
               uint256 reservePrice) public pure returns(uint256){

      uint256 y1 = initailPrice - reservePrice;
      return  (elapsedTime*(-y1) + y1*totalTime)/totalTime;
  }

}


contract InverseLogarithmicDecreasingStrategy is IDecreasingStrategy{

  function getCurrentPrice(uint256 elapsedTime,
               uint256 totalTime,
               uint256 initailPrice,
               uint256 reservePrice) public pure returns(uint256){
      uint256 y2 = initailPrice - reservePrice;
      uint256 a = y2/log2(totalTime + 1);

      uint256 tmpval = a*log2(elapsedTime + 1);
      return y2 - tmpval;
  }

}
```

```
74
75 contract LogarithmicDecreasingStrategy is IDecreasingStrategy{
76
77   function getCurrentPrice(uint256 elapsedTime,
78               uint256 totalTime,
79               uint256 initailPrice,
80               uint256 reservePrice) public pure returns(uint256){
81       uint256 y2 = initailPrice - reservePrice;
82       uint256 a = y2/log2(totalTime + 1);
83
84       return a*log2(totalTime - elapsedTime + 1);
85
86   }
87
88
89 }
```

5 Decreasing Strategies Tests

```
pragma solidity >=0.4.22 <0.6.0;
import "remix_tests.sol"; // this import is automatically injected by Remix.
import "./DecreasingStrategies.sol";


contract test_1 {
  IDecreasingStrategy dslinear;
  IDecreasingStrategy dslog;
  IDecreasingStrategy dsinverselog;
  function beforeAll() public {
    dslinear = new LinearDecreasingStrategy();
    dslog = new LogarithmicDecreasingStrategy();
    dsinverselog = new InverseLogarithmicDecreasingStrategy();

  }

  function check1_linear() public returns(bool) {
    // use 'Assert' to test the contract

    uint256  res1  = dslinear.getCurrentPrice(1,4,4,0);
    Assert.equal(res1, 3, "res 1 not working");
    uint256  res2  = dslinear.getCurrentPrice(3,4,4,0);
    Assert.equal(res2, 1, "res 2 not working");
    uint256  res3  = dslinear.getCurrentPrice(2,4,4,0);
    Assert.equal(res3, 2, "res 3 not working");

  }

  function check2_linear() public returns(bool) {
    uint256  res1  = dslinear.getCurrentPrice(2,8,4,0);
    Assert.equal(res1, 3, "res 1 not working");
    uint256  res2  = dslinear.getCurrentPrice(6,8,4,0);
    Assert.equal(res2, 1, "res 2 not working");
    uint256  res3  = dslinear.getCurrentPrice(4,8,4,0);
    Assert.equal(res3, 2, "res 3 not working");

  }

  function check3_log() public returns(bool){
    uint256 res0 = dsinverselog.getCurrentPrice(0,8,4,0);
    Assert.equal(res0, 4, "res 1 not working");
    uint256 res1 = dslog.getCurrentPrice(4,8,4,0);
    Assert.equal(res1, 3, "res 1 not working");
    uint256 res2 = dslog.getCurrentPrice(2,8,4,0);
    Assert.equal(res2, 3, "res 2 not working");
    uint256 res3 = dslog.getCurrentPrice(6,8,4,0);
    Assert.equal(res3, 2, "res 3 not working");
    uint256 res4 = dslog.getCurrentPrice(8,8,4,0);
    Assert.equal(res4, 0, "res 3 not working");
  }

  function check4_inverselog() public returns(bool){
    uint256 res0 = dsinverselog.getCurrentPrice(0,8,4,0);
    Assert.equal(res0, 4, "res 0 not working");
    uint256 res1 = dsinverselog.getCurrentPrice(4,8,4,0);
    Assert.equal(res1, 1, "res 1 not working");
    uint256 res2 = dsinverselog.getCurrentPrice(2,8,4,0);
    Assert.equal(res2, 2, "res 2 not working");
    uint256 res3 = dsinverselog.getCurrentPrice(6,8,4,0);
    Assert.equal(res3, 1, "res 3 not working");
    uint256 res4 = dsinverselog.getCurrentPrice(8,8,4,0);
    Assert.equal(res4, 0, "res 4 not working");
  }
}
```