Science and Technology Council, Indian Institute of Technology Kanpur

Game of Blocks

Project Report

July 28, 2021

Contents

1 Overview

This project aims to introduce the concept of blockchains and smart contract development, which have recently become quite prevelant and relevant, with a flavor of Game Theory.

1.1 Goals Achieved

1. Understanding basic mechanism of Blockchains and cryptocurrencies.
2. Implementing a simple mining algorithm, in the form of an Assignment.
3. Learning about various consensus mechanisms prevalent in Blockchain systems.
4. Understanding Ethereum Smart Contracts and their usage.
5. Learning the programming language Solidity.
6. Implementing a Smart contract for transfer of a self created token called MetaCoin to settle loans, in the form of an Assignment.
7. Learning the basics of Game Theory and Mechanism Design, and the different types of mechanisms for various tasks.
8. Understanding the importance of designing fool proof mechanisms for day-to-day games.
9. Realizing the need for decentralization in everyday games (tasks) such as auctions, voting and asset transfer.
10. Implementing several different methods to perform these tasks in a decentralized fashion, using Smart Contracts on the Ethereum Blockchain.

2 Concept of Blockchain

2.1 History and Origin

Cryptographer David Chaum was the first to propose a blockchain-like protocol in 1982. Stuart Haber and W. Scott Stornetta worked further on making a it cryptographically secure and immutable. The first conceptualization and implementation however was done as late as 2008 by an anonymous group/person under the name of Satoshi Nakomoto. Nakomoto created the first cryptocurrency- bitcoin and created the first blockchain database(ledger system) in the process. Nakomoto also authored the famous bitcoin white paper which inspired the term 'Blockchain'. This concept of blockchains has been put to use to develop other decentralised, cryptographically secure applications. One of the most versatile ways of using this technology is through Smart Contracts in the Ethereum ecosystem. Ethereum blockchain has been the most versalite, emerging blockchain that is set to revolutionise finance as we know it.

2.2 What is a Blockchain?

A blockchain database is quite literally blocks of data connected by links(chains) and stored in a decentralised manner in the network. The links are secured using cryptographic hash functions(like SHA-256), thus making it impossible to change the transactions/blocks on the chain without redoing the hash-based proof-of-work. As there is no centralised authority directing funds and maintaining accounts, there could be a double-spending problem which is prevented by validation of a transaction by other nodes according to the proof-of-work consensus. This makes the blockchain transactions strictly peer-to-peer, thus enabling complete privacy and confidentiality. Messages(events) are broadcast into the network and the nodes compete to validate the block. In bitcoin,The longest proof-of-work chain is taken to be the correct one. Ethereum on the other hand, is trying to move to a Proof-of-Stake consensus mechanism because the computation power necessary to mine a block is increasing with more miners added to the network. It is worth noting that the term blockchain was never mentioned in the bitcoin white paper. The name is quite trivial though considering the structure and implementation of the database system.

3 What are Blocks?

Transactions are passed from node to node within the network, so the order in which two transactions reach each node can be different. How do you know which transaction has been requested first? It's not secure to order the transactions by timestamp because it could easily be counterfeit. Therefore, there is no way to tell if a transaction happened before another, and this opens up the potential for fraud. If this happens, there will be disagreement among the network nodes regarding the order of transactions each of them received. So the blockchain system has been designed to use node agreement to order transactions and prevent the fraud described above. Blocks in the blockchain network are a collection of a definite number of transactions that happened in the network. Blocks are therefore organized into a time-related chain. Transactions in the same block are considered to have happened at the same time, and transactions not yet in a block are considered unconfirmed. This is what puts one block after the other in time. Blocks are broadcast into the network to be validated by miners by computing a random number which when appended to the block and hashed produces a fixed number of zeroes (which is mathematically calibrated periodically) to ensure the number of blocks mined over time is constant. This is done to ensure there is no confusion about which block was created and validated first.

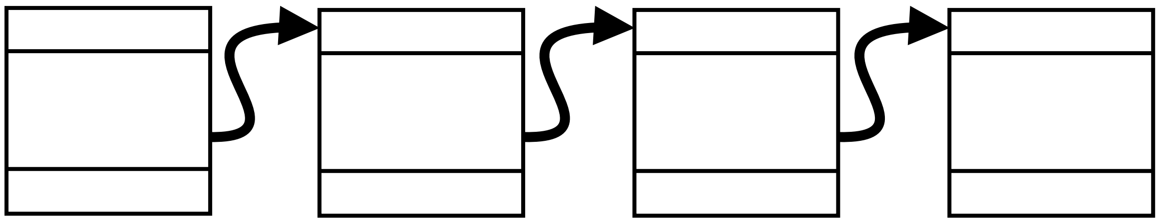


Figure 1: Blockchain

So, the blocks in the blockchain network basically represents the validated subunits that are added to the ledger file that is stored in every node in the network. The transactions are stored in the form of a Merkle Tree so that spent transactions can be safely discarded without breaking the block's hash, to free disk space. Thus, the blockchain database is essentially a chain of blocks with each block containing the transactions stored in the form of a Merkle Tree so that the older blocks can later be compacted.

4 Hashing

Hashing is what makes blockchains **cryptographically secure**. A hash function takes any arbitrary data as input and maps it to a fixed size output. The mapping is expected to be perfectly even with respect to the range and must not be predictable in anyway. This makes it virtually impossible to find the possible arguments that could have been passed to the function given the output. This makes hash functions very useful in the proof-of-work consensus mechanism to ensure that maximum CPU power is used to work and find the nonce of a given block. The nonce computed is random because of the nature of the hash function and thus difficulty can be altered quite easily to ensure the average number of blocks mined in a given time doesn't change in the blockchain network. Verifying the hash is very easy and requires minimal CPU power and so the mined blocks can easily be verified by the other nodes.
Hashing is also used to digitally sign transactions on a coin when the coin is transferred to the payee. Each owner transfers the coin to the next by digitally signing a hash of the previous transaction and the public key of the next owner and adding these to the end of the coin. A payee can verify the signatures to verify the chain of ownership. Hashing is also done by timestamp servers as computational proof of the chronological order of the transactions

5 Peer to Peer Nature of Blockchains

Probably, the most alluring part of blockchain networks is their pure peer-to-peer nature. This enables secure transactions without the need of a central authority. Thus, the transactions happen without the need for trust in someone directing the funds and maintaining accounts. This aspect of blockchains is what makes **smart contracts** especially attractive.

The P2P nature of blockchains is particularly useful in solving the double-spending problem. The problem of double spending is solved by maintaining a peer-to-peer distributed timestamp server. This server generates computational proof of the chronological order of transactions. In a P2P network, blocks have to be validated and this job is done by the miners in the network and rewards are provided for mining to incentivize the process. The users' personal information is confidential and irrelevant, contrary to what happens in a centralised network such as a bank. Though this guarantees privacy to the users, this can also lead to illegal activity on the network. So, there is hesitation among all the governments around the world in adopting cryptocurrencies as a legally accepted form of money. On the other hand, the cost to perform a value transaction from and to anywhere on the planet is very low. This allows micropayments.Value can be transferred in a few minutes, and the transaction can be considered secure after a few hours. Anyone can verify the transactions, making the technology fast and transparent.

6 Consensus algorithm

We know that Blockchain is a decentralized network that provides privacy, security, and transparency. There is no central authority present to validate and verify the transactions, yet every transaction in the Blockchain is considered to be completely secured and verified. This is possible only because of the presence of the **consensus protocol** which is a core part of any Blockchain network.
A **consensus algorithm** is a procedure through which everyone on the blockchain network is able to reach an agreement on the current state of the distributed ledger. Essentially, the consensus protocol ensures that every new block added to the blockchain is the **only** true block, that is agreed upon by all the nodes in the network. Two of the most common consensus algorithms are Proof of Work (POW) and Proof of Stake (POS).

6.1 PoW vs PoS

The idea for Proof of Work (PoW) was first published in 1993 by Cynthia Dwork and Moni Naor and was later applied by Satoshi Nakamoto in the Bitcoin paper in 2008. Proof of Work consensus is the mechanism of choice for the majority of cryptocurrencies currently in circulation. The term "proof of work" was first used by Markus Jakobsson and Ari Juels in a publication in 1999.
The Proof of Work consensus algorithm involves solving a **computational challenging puzzle** in order to create new blocks in the blockchain. This is usally finding an appropriate nonce value such that the hash of the block and the nonce is less than a pariticular target value set by the blockchain nodes. Colloquially, the process is known as 'mining', and the nodes in the network that engage in mining are known as 'miners'. The incentive for mining transactions lies in economic payoffs. For example, on the bitcoin blockchain, competing miners are rewarded with 6.25 bitcoins (this number is halved every few years. It was 50 initially in 2009.) and a small transaction fee.

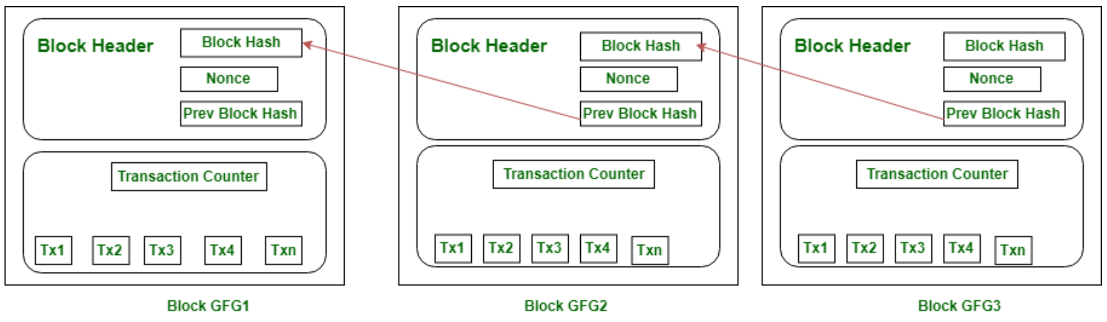


Figure - Proof of Work

Proof of Stake (PoS) is the most common alternative to PoW. Ethereum will soon shift from PoW to PoS consensus. In this type of consensus algorithm, instead of investing in expensive hardware to solve a complex puzzle, validators textbfinvest in the coins of the system by locking up some of their coins as stake. After that, all the validators will start validating the blocks. Validators will validate blocks by placing a bet on it if they discover a block which they think can be added to the chain. Based on the actual blocks added in the Blockchain, all the validators get a **reward proportionate to their bets** and their stake increase accordingly.
In the end, a validator is chosen to generate a new block based on their economic stake in the network. Thus, PoS encourages validators through an incentive mechanism to reach to an agreement.

7 Vulnerabilities

Despite all the security that blockchain technology provides, cryptocurrencies are not a 100% secure. Over the years, there have been a few major cases of large amounts of cryptocurrencies getting stolen. Most of these have been cases of phishing, weak protection of employee login credentials and not using multisignature private keys - which are all mistakes which cannot be directly attributed to blockchain technology. It is often argued that blockchain transactions are highly secure because they are recorded on an allegedly immutable record. However, each transaction has a signature and the signature may be manipulated before the closure of the transaction. An example of this is the theft of approximately 850,000 bitcoins from Mt. Gox, which until 2014 was handling 70% of all all BTC transactions worldwide. One of the largest attacks in the history of cryptocurrencies, it was conducted by hackers who submitted code changes to a public ledger before the posting of the initial transactions.

8 Forks in Blockchain

A fork in a Blockchain is essentially a split in the blockchain network. Occasionally, more than one block is added to the blockchain by miners at the same time. This creates a fork in the chain. In such an instance, the nodes will always consider working with the longest chain, and the shortest chain is abandoned. Note that this doesn't mean that the transactions on the abandoned block are discarded, they are simply added to the rest of the transactions waiting to be verified by the miners again.

The instance described above was that of an *Accidental fork.* But not all forks are accidental and may even result in a permanent split. Like, in case of a software update, or changing the consensus protocol, where not all the nodes agree to the changes. These are instances of *Intentional forks.* Intentional forks can also be classified further into *hard* and *soft* forks.

8.1 Hard Forks

A hard fork occurs when nodes of the newest version of a blockchain no longer accept the older versions of the blockchain, which creates a permanent divergence from the previous version of the blockchain. This typically occurs when there is a change of consensus rules (i.e. block size, mining algorithm, consensus protocol) in a way that makes the previous versions of the of the blockchain incompatible. This fork creates two paths, one that follows the updated set of rules and the other continues along the old path. Generally, the majority of the nodes agrees to update and after a short time, those on the old chain realize that their version of the blockchain is outdated or irrelevant and quickly upgrade to the latest version.

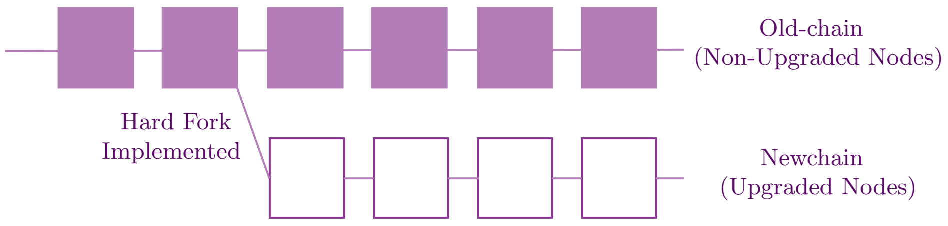


Figure 2: Hard Fork

But, not all such forks are unanimously agreed upon. When a significant portion of the full nodes disagree upon the changes to the blockchain, the two disagreeing factions will fork the chain and implement the changes they desire on their respective chains. In most cases, this results in the formation of a new type of cryptocurrency. Like in 2014, a hard fork in the Ethereum blockchain resulted in a split, creating Ethereum (ETH) and Ethereum Classic (ETC) chains. A more recent example is of Bitcoin in 2017, which resulted in a split creating Bitcoin Cash.

8.2 Soft Forks

A soft fork occurs when *backwards compatible* changes are introduced in the consensus rules. Soft forks do not need network nodes to upgrade, because all blocks on the soft-forked blockchain follow the old set of consensus laws in addition to the new ones. It is much easier to implement a soft fork as this requires only a majority of the miners upgrading to enforce the new rules, as opposed to a hard fork that requires all nodes to upgrade and agree on the new version.
Non-upgraded nodes will still continue to see that the incoming new transactions are valid. The issue is when non-upgraded miners try to mine new blocks. Initially, when there are enough non-upgraded miners, their blocks will be accepted by the network. But eventually, when more than 51% of the hashing power moves to the updated soft fork, the blocks mined by the non-upgraded miners will be rejected by the network. Hence, soft forks represent a gradual upgrading mechanism as those who have yet to upgrade their software are incentivized to do so, or risk having reduced functionalities. An example of a soft fork is the Pay

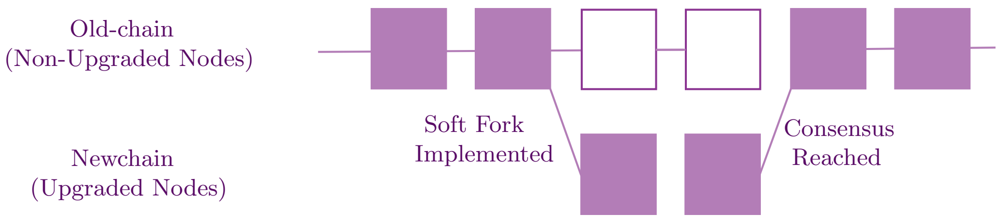


Figure 3: Soft Fork

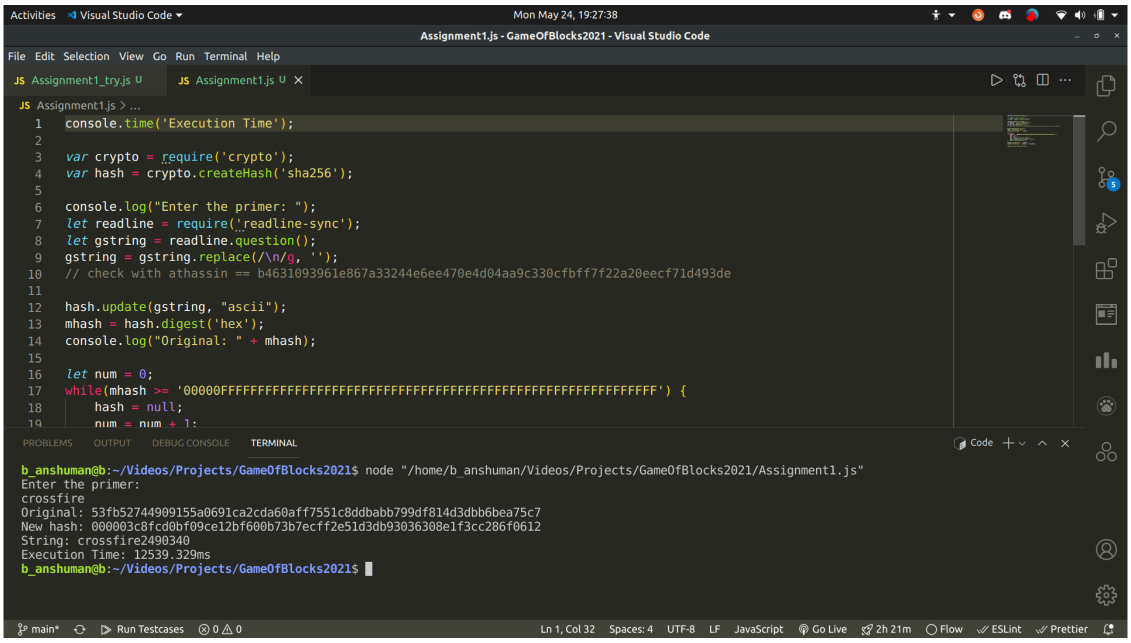
to Script Hash (P2SH) soft fork which generated multi-signature addresses on the Bitcoin network.

9 Assignment : Basic Implementation of Mining

Aim : User has to input a string, whose hash is displayed, and the program tries to concatenate a number, trying each time to get the nounce, the digest, less than the target value, which is:

0x00000FFFFFFFFFFFFFFFFFF.......FFFFFFFFFFFFFFFF

This involved finding the nounce of a string, and concatenating a number and rechecking the nounce to get it below a required value. Implemented with the help of node and using crypto library of Js.



```javascript
console.time('Execution Time');

var crypto = require('crypto');
var hash = crypto.createHash('sha256');

console.log("Enter the primer: ");
let readline = require('readline-sync');
let gstring = readline.question();
gstring = gstring.replace(/\n/g, '');
// check with athassin == b4631093961e867a33244e6ee470e4d04aa9c330cfbff7f22a20eecf71d493de

hash.update(gstring, "ascii");
mhash = hash.digest('hex');
console.log("Original: " + mhash);

let num = 0;
while(mhash >= '00000FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF') {
    hash = null;
    num = num + 1;
```

```
b_anshuman@b:~/Videos/Projects/GameOfBlocks2021$ node "/home/b_anshuman/Videos/Projects/GameOfBlocks2021/Assignment1.js"
Enter the primer:
crossfire
Original: 53fb52744909155a0691ca2cda60aff7551c8ddbabb799df814d3dbb6bea75c7
New hash: 000003c8fcd0bf09ce12bf600b73b7ecff2e51d3db93036308e1f3cc286f0612
String: crossfire2490340
Execution Time: 12539.329ms
b_anshuman@b:~/Videos/Projects/GameOfBlocks2021$
```

The main idea was hashing concept, that we have to iterate through all possibility before we arrive at an apt nounce. We had to keep track of time, for which vanilla Js '`console.time()`' function was enough.

10 Introduction to Ethereum

First of all, AltCoins are all cryptocurrencies other than BitCoin.
Why do we need any **AltCoins**, if BitCoin is working fine? The answer is quite wide of course, but the basic idea of cryptocurrencies, and blockchains in general is the idea of decentralization.

Introduction of more such crypto-currencies, have enforced this underlying principle of blockchains. Also, BitCoin still inherits flaws that couldn't be predicted by its anonymous creator Satoshi Nakamoto, including being energy-inefficient and inflexible. Now moving on to the point of discussion.

Ethereum is a **decentralized, open-source blockchain with smart contract functionality.** ETH (ether) is the actual cryptocurrency involved in this blockchain system.
Ethereum is the most active blockchain in the world, currently. It is actively used to create and host decentralized apps, or Dapps, which actually are *Smart Contracts*. Its creator are Vitalik Buterin and Gavin Wood, and this was initially community funded.

The idea of Ethereum was mainly to implement the smart contract system, and The platform allows developers to build and operate decentralized applications that any user can interact with. It has become a battle-tested platform on which other tokens can build-over.

There are some standards over which smart contracts started getting standardised:

- ERC-20 Fungible Tokens
- ERC-721 Non-fungible Tokens.
- ERC-1155 Best of both standards.

Fungiblity refers to being like a denomination of a currency. All tokens of the same denomination accounts for the same value. Non-fungible therefore refers to each token being unique on its own, and this type of token is useful to assign value to art-forms and physical entities. This standardisation was necessary and a community decision, cause of rampant smart contracts surfacing each day, with no good means to compare to each other.

Another great thing about Ethereum over BitCoin is, its going to implement proof of stake in *Ethereum 2.0*, which is far more efficient consensus, but not much hardened against exploitations.

11 Smart Contracts

We need to talk about this, before moving on.
Smart contracts, as the term suggests, are literally contracts, a piece of code, program, which is instructed beforehand to perform something once some pre-planned conditions are

met, and the main idea is that the implementation itself requires no third-party. The contract decides itself, when to execute, and its execution is transmitted everywhere.

More formally, Smart contracts are **event-driven code contracts with state attributes**, used to automate the execution of a pre-made-agreement so that all participants can be immediately certain of the outcome, without intermediary involvement or time delay.

Such guarantees are there because of this is implemented on a blockchain. The blockchain itself is updated when this transaction takes place, and therefore no-one can deny its happening, and parties who are granted permission can see the results.

A layman's analogy of a smart contract is a vending machine. Insert a coin (feedback), the vending machine was designed (programmed) such as it would deliver (transactions) your drinks (assets) according to the feedback, without anyone's intervention.

12 Ethereum Virtual Machine

This concept has transformed what cryptocurrencies are meant for. It takes the idea of a blockchain, beyond the ledger meaning, and enabling to take decisions, given some initial code and feedback. This decision, once all requirements are achieved, is fail-safe, literally. Therefore Ethereum is called a distributed state machine, not just a decentralised ledger, that just tracks transactions.

A state machine/finite state machine is, abstractly and in mathematical terms, is a model that can be in exactly one of a finite number of states at any given time. Fixed and unique **state** at a time, i.e. its configurations but in a broader sense.

A state machine can change from one state to another in response to some inputs; the change from one state to another is called a *transition*. Such a model is completely defined by a list of its states, its initial state, and the inputs that trigger each transition.

Ethereum's state is a large data structure which holds not only all accounts and balances, but a machine state, which can change from block to block according to a pre-defined set of rules, and which can execute arbitrary machine code. Ethereum Virtual Machine(referred to as EVMs from now), therefore being a State Machine, that too decentralised, therefore has some remarkable features.

- Can execute without failing if the pre-requisites are met.
- Extremely flexible as can include arbitrary code.
- Can't be down due to any kind of local problems, because of decentralisation.
- All costs and value circulation are confined to this blockchain itself, value being transacted by ETH. So a self-reliant system.

This feature of Ethereum, being a state machine, enables smart contract implementation, in a practical manner. As mentioned prior, Smart contracts intrinsically contain state attributes Therefore being deployed on a state machine, they are quite reliable. Let's compare to a case when a smart contract is deployed on a regular network (still decentralised). The code would still be available to everyone, but because there is no concept of 'state', there is always a possiblility to reverse the decision and execution of the contract, cause the network doesn't know if the execution had taken place. Note that I am talking about the network itself, not the nodes. Therefore Ethereum is unique, needed as a blockchain, and as a platform that can manage smart contracts.

13 What is Gas in Ethereum?

Whenever you run a smart contract on the Ethereum network (EVM), certain amount of storage and computational power is used. When miners on the network perform the required computational work and verify these transactions, what incentivizes them to do so? Who pays them and how much? This is where *gas* in Ethereum comes in. 'Gas' refers to the unit that measures the amount of computational effort required to execute specific operations on the Ethereum network. Each operation on the EVM consumes gas. For example, a multiplication (MUL) consumes 5 gas and an addition (ADD) consumes 3 gas. More details about these operations and their gas consumption can be found in the Ethereum yellow paper.

13.1 Gas Price

For every transaction a user makes, he has to pay a fee to the miner. The amount of fees (in ETH, usually denoted in gwei) required for one unit of gas is reffered to as the *gas price*. While every operation in the EVM consumes a predefined amount of gas, a user can specify a gas price in every transaction. This is discussed later. The current default gas price is 0.2μ ethers (or 0.000002 ETH).

13.2 Gas Limit

Gas limit refers to the maximum amount of gas you are willing to consume on a transaction. More complicated transactions, involving smart contracts, require more computational work so they require a higher gas limit than a simple payment. A standard ETH transfer requires a gas limit of 21,000 units of gas. A user has to specify a gas limit in every transaction. On the EVM, gas is consumed with each operation. If all the gas consumed reaches the gas limit without the transaction being completed, an *Out of Gas* exception occurs. In such an instance, all the operations that were performed and state changes are reverted, the user pays all the gas fees corresponding to all the work performed by the miner, and the failed transaction is included in a block. In case the gas limit is higher than the total amount of gas consumed by the transaction, the excess gas is refunded.

13.3 Gas vs Price

A transaction with a very low gas limit will not even reach the miners, regardless of the gas price. If the gas limit is adequate for a transaction, but the fee is too low, even though the transaction may reach miners, they won't process it. Fees determine the order in which the transactions are added in the blockchain. Since an Out of Gas exception is practically a waste of money for a user, it may seem as if providing a high gas limit seals the deal. But this is not the case. A miner only gets paid for the actual gas consumed by the transaction, and any excess gas is refunded to the user. Thus, miners are likely to prioritize a set of "small" transactions over a single transaction with a very high gas limit. So setting a very high gas limit can lead to delay before the transaction is eventually added to a block.

14 Solidity

Another programming language! One might ask, why do we need this, as Solidity is often described as a child language of JavaScript, C and Python. Take a look at the official documentation.

"Solidity is a statically-typed curly-braces programming language designed for developing smart contracts that run on the Ethereum Virtual Machine. Smart contracts are programs that are executed inside a peer-to-peer network where nobody has special authority over the execution."

Solidity was created for implementing smart contracts in the first place, that too efficiently, securely and syntactically flexible as can be. The field of smart contracts is sensitive, handling billions worth of value, and strict measures have to be taken, and being secure is the priority. This led to a few uncommon features about solidity (found while doing this project):

14.1 No Back-Compatibility

The first thing one would encounter while being introduced to Solidity is:

```
pragma solidity ^0.8.0
```

We are declaring compiler version with which the contract in context is compatible with. Why? Its not common, at least wasn't for me. The reason is the topic. All compilers are updated, new versions are launched, but most compilers support their previously handled commands, which might have been deprecated. But that leads to inheriting previous flaws and exceptions as well. This can't be done in case of smart contracts. Any vulnerability, is a vulnerability, can be exploited some way or the other.

Therefore while creating your contract you have to designate the compatible compilers which your code can run on, and it is recommended to go for the latest compiler, unless you know what you are doing.

14.2 No Support for Floating numbers

Vanilla Solidity doesn't support floating point! There are some external libraries which can implement that, but none of them is official. Strange? π is literally 3 here, natively.

This is from the updated documentation:"Fixed point numbers are not fully supported by Solidity yet. They can be declared, but cannot be assigned to or from."

But of course you can work-around this *minor* drawback. Actually, EVM itself doesn't support floating points. You will have to understand floats implementation to understand why, here is the link. For the workarounds, this dialogue explains how to concisely:

BOB: Please EVM, $\frac{3}{2}$ =?
EVM: Lol, 1.

BOB: Ok... what about $\frac{3000}{2}$ =?
EVM: Easy; 1500.

See? Manipulating numbers a bit would manage most of the calculations that one would be needed to perform, satisfactorily. You multiply by how many decimals you want to be precise to first, e.g. 10^{18} do your calculation, then divide to remove the extra decimals. It's more precise and better for financial calculations

14.3 Several Numeric Types

Precisely, 5,248. Yes, according to the documentation, there are 32 signed integer, 32 unsigned integer, 2592 signed fixed-point, and 2592 unsigned fixed-point types. JavaScript has only two numeric types. Python 2 used to have four, but in Python 3 type "long" was dropped, so now there are only three. Java has seven and C++ has something about fourteen.

So many data types, and still can't support floating points. This has a reason. Common types used in other languages are quite flexible, but inherit weird semantics of underlying CPU instructions, like silent over- and underflow, asymmetric range, binary fractions, byte ordering issues etc. Straightforward usage makes the program vulnerable, and secure usage renders it unreadable.

Actually, EVM natively supports two data-types, 256-bit word and 8-bit byte. Therefore it supports 2 numeric types originally, signed 256-bit and unsigned 256-bit integers. All data-types of int in solidity are just multiples of 8. Assigning so many different spaced numeric types made this more flexible, as we have to save space and computational power consumed, as much as possible.

14.4 Usual Arithmetic avoided

Solidity does support all usual arithmetic symbols, but there use for calculations are not recommended. This is unusual, but again, needed for being more secure.

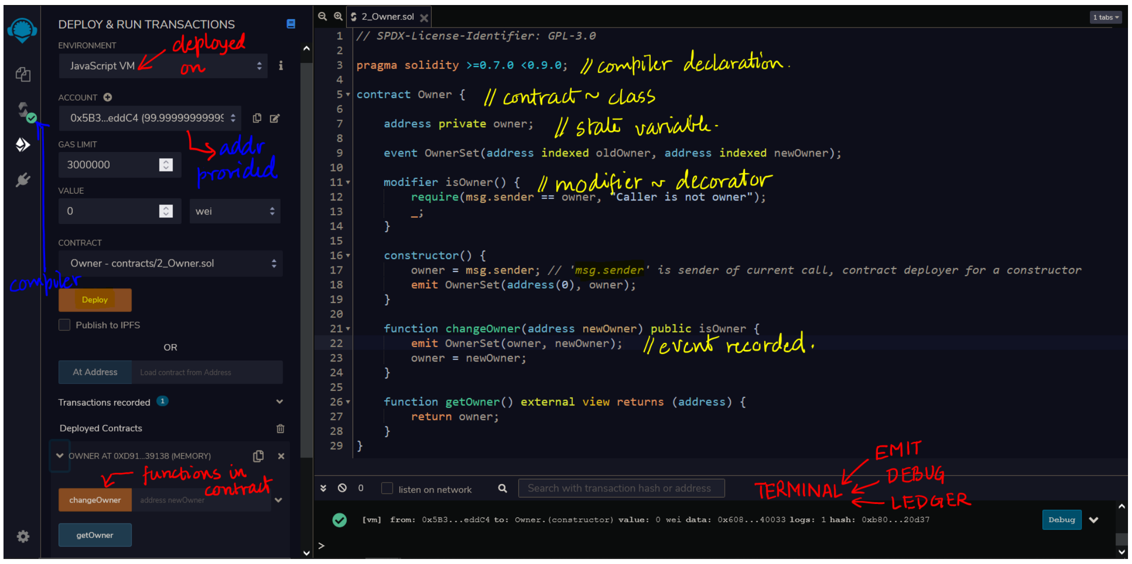
There is a 'Safemath' Library in Solidity, whose source code you can look up right here. Its implementation is trivial, nearly too trivial to be thought as needed.

Arithmetic operations in Solidity wrap on overflow. This can easily result in bugs, because programmers usually assume that an overflow raises an error, which is the standard behavior in high level programming languages. 'SafeMath' restores this intuition by reverting the transaction when an operation overflows. Using this library instead of the unchecked operations eliminates an entire class of bugs, so it's recommended to use it always.

This list can maybe go on, but these points are what I came across during this project. Else, Solidity is a neat language, actually quite similar to Javascript, and extremely intuitive keywords.

For getting started with Solidity, which is a language, for smart contracts, you would require a network. Your contract may be deployed directly to the official Ethereum Virtual Machine, which would cost you ETH and Gas, but as a beginner, one is recommended to use REMIX, an online IDE made for the development of Smart Contracts. You can move directly to local compilers that deploy your contract to popular public test networks: Ropsten, Göerli, Kovan, and Rinkeby. All of these networks can be accessed via Infura's API.

We developed basic smart contracts through REMIX, so would explain about that.


```
DEPLOY & RUN TRANSACTIONS
ENVIRONMENT      JavaScript VM
ACCOUNT          0x5B3...eddC4 (99.9999999999
GAS LIMIT        3000000
VALUE            0  wei
CONTRACT         Owner - contracts/2_Owner.sol
[Deploy]
[ ] Publish to IPFS
OR
[At Address]  Load contract from Address
Transactions recorded 1
Deployed Contracts
OWNER AT 0XD91...39138 (MEMORY)
[changeOwner]  address newOwner
[getOwner]
```

```
2_Owner.sol

// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.7.0 <0.9.0;

contract Owner {

    address private owner;

    event OwnerSet(address indexed oldOwner, address indexed newOwner);

    modifier isOwner() {
        require(msg.sender == owner, "Caller is not owner");
        _;
    }

    constructor() {
        owner = msg.sender; // 'msg.sender' is sender of current call, contract deployer for a constructor
        emit OwnerSet(address(0), owner);
    }

    function changeOwner(address newOwner) public isOwner {
        emit OwnerSet(owner, newOwner);
        owner = newOwner;
    }

    function getOwner() external view returns (address) {
        return owner;
    }
}
```

```
0  listen on network   Search with transaction hash or address
[vm] from: 0x5B3...eddC4 to: Owner.(constructor) value: 0 wei data: 0x608...40033 logs: 1 hash: 0xb80...20d37   Debug
>
```

Default Layout in Remix IDE

Pardon my squiggles. The above picture briefly explains what remix is all about, as a beginner. Write your contract, syntax is intuitive as said before, compile, then choose your virtual environment on which you want to deploy this on (in this case, JavaScript VM), choose one of the 15 addresses Remix provides by default, then Deploy, and interact with the contract by exchanging addresses, taking care of the Debug terminal at the bottom.

Note the License declaration at the top of the .sol file, the extension of solidity files.

From Solidityˆ 0.6.8 SPDX License is introduced, which needs to be in a contract before being deployed.

You're good to get started with Solidity!

15 Assignment : Basic Contracts with Solidity

Implementation of a basic set of contracts on REMIX IDE.
Task was to initialise an entity 'Owner' with 1,00,000 credits, or *Meta Coins* (the name of token implemented), who has acquired this many credits by taking loans from multiple users/addresses.

15.1 Contract MetaCoin

This Contract defines our currency. Includes '`sendCoin`' and '`getCoin`'.

- `sendCoin(none)` - to send MetaCoin.
- `getCoin(none)` - to check the balance of that addr.

15.2 Contract Loan

A smart contract, that includes functions:

- `getCompoundInterest(pure)` - returns an approximate value of compound interest given P, R, T.
- `getOwnerBalance(view)` - returns Owner's balance, and is accessible by all.
- `reqLoan(none)` - to request repayment of loan taken by the Owner, by specifying P, R, T and is broadcasted.

that can be accessed by all entities in the system, and the functions:

- `viewDues(view)` - to view Dues of a particular creditor.
- `settleDues(none)` - to settle Dues with that creditor provided Owner has enough balance.

accessible only by the owner, to settle his debt.
Two more auxiliary functions used were:

- `add(pure)` - to add more securely.
- `mulDiv(pure)` - to implement this arithmetic more securely.

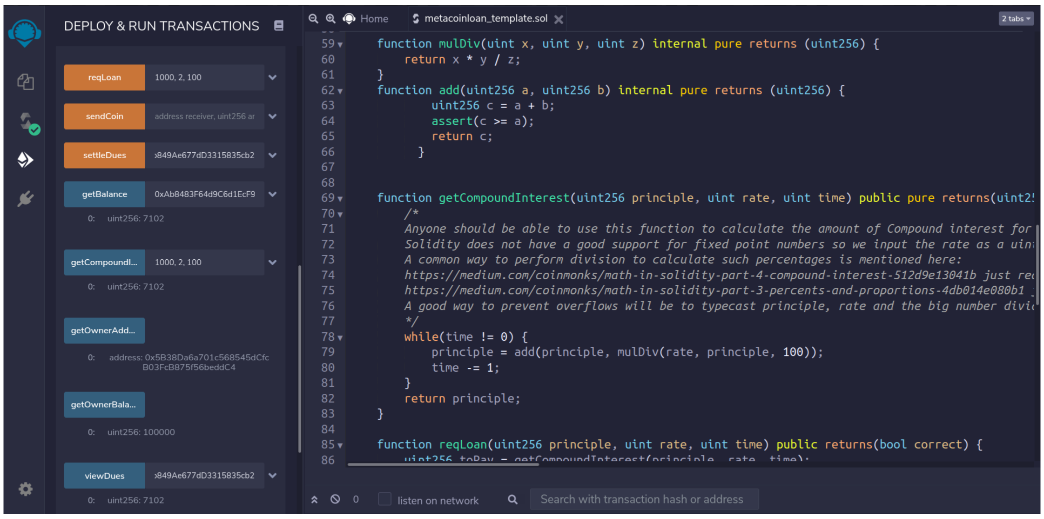
These contracts can be easily run on REMIX by deploying on a JavaScript Virtual Machine, with default parameters being enough to get started.

15.3 Summary

Take a scenario in which address '0x5B38Da6a701c568545dCfcB03FcB875f56beddC4' is the owner and '0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2' is the creditor.

Deploying the *Loans* contract initiated the current active address as the owner, with the above number of tokens via the constructor of *MetaCoin* contract. The creditor then requested for repayment via '`reqLoan`' function, which was then viewed by the owner, and settled through '`viewDues`' and '`settleDues`' respectively.

Following is the screenshot of the above scenario.


DEPLOY & RUN TRANSACTIONS
Home
metacoinloan_template.sol
2 tabs
listen on network
Search with transaction hash or address

```
reqLoan          1000, 2, 100
sendCoin         address receiver, uint256 ar
settleDues       ›849Ae677dD3315835cb2
getBalance       0xAb8483F64d9C6d1EcF9
    0:   uint256: 7102
getCompoundI...  1000, 2, 100
    0:   uint256: 7102
getOwnerAdd...
    0:   address: 0x5B38Da6a701c568545dCfc
                  B03FcB875f56beddC4
getOwnerBala...
    0:   uint256: 100000
viewDues         ›849Ae677dD3315835cb2
    0:   uint256: 7102
```

```
    function mulDiv(uint x, uint y, uint z) internal pure returns (uint256) {
        return x * y / z;
    }
    function add(uint256 a, uint256 b) internal pure returns (uint256) {
            uint256 c = a + b;
            assert(c >= a);
            return c;
          }


    function getCompoundInterest(uint256 principle, uint rate, uint time) public pure returns(uint2
        /*
        Anyone should be able to use this function to calculate the amount of Compound interest for
        Solidity does not have a good support for fixed point numbers so we input the rate as a uin
        A common way to perform division to calculate such percentages is mentioned here:
        https://medium.com/coinmonks/math-in-solidity-part-4-compound-interest-512d9e13041b just re
        https://medium.com/coinmonks/math-in-solidity-part-3-percents-and-proportions-4db014e080b1
        A good way to prevent overflows will be to typecast principle, rate and the big number divi
        */
        while(time != 0) {
            principle = add(principle, mulDiv(rate, principle, 100));
            time -= 1;
        }
        return principle;
    }

    function reqLoan(uint256 principle, uint rate, uint time) public returns(bool correct) {
        uint256 toPay = getCompoundInterest(principle, rate, time);
```

15.4 A Subtle Warning

The Compound Interest Implementation in these contracts is inherently flawed.
This is because of the fact that Solidity at its current stage, doesn't support floating-point numbers. Therefore though acceptable for large transactions, for smaller values, value returned by '`getCompoundInterest`' function malfunctions.

This can be avoided by another algorithm for Compound Interest calculation.
Example of error-prone inputs:

- `getCompoundInterest(10, 2, 100)`
  would return 10, as though no interest has been applied.
- `getCompoundInterest(100, 2, 10)`
  would return 120, just as though of a simple interest calculation.

This is happens cause of neglecting the small increase of principle per iteration.

15.5 Updating the Loan Contract

To implement a function for retrieving the address of the creditor with max request. This function can only be accessed by the owner.

Two implementations, one by modifying the original file, and is more gas efficient, but has a critical drawback.
The second implementation is through loops and mapping. Which is more reliable, but gas consumption peaks.

First Method:

As in the main, 'Assignment2.sol' file, the function '`getMaxAddress()`' (view) is implemented by keeping 2 state variables, '`maxDebt`' and '`maxDebtAddress`' which are updated everytime someone requests for a repayment. The critical flaw in this implementation is, if the owner repays the max Creditor, we won't have any idea of how to point to the new max creditor.

Second Method:

Showed in the ‘Assignment2_update.sol’ the function ‘`getMaxAddress()`’ (view) is implemented by looping through the mapping of creditors, while the mapping is updated every time someone claims for a repayment through ‘`reqLoan`’ function. This implementation has no such flaw as above, but is extremely gas-inefficient, as the looping is done throughout the mapping every time the owner requests for ‘`getMaxAddress()`’.

16 Game Theory

Game theory is the study of mathematical models of strategic interaction among rational decision-makers. It has applications in all fields of social science, as well as in logic, systems science and computer science.
One of the first and most fundamental applications of Game Theory was to zero-sum games. A zero-sum game is a mathematical representation of a situation in which an advantage that is won by one of two sides is lost by the other. A classical example is the Prisoner's Dilemma. The following example explains this game and how game theory dictates what a player should do in that situation:
The simplified game is as follows:

> Two members of a criminal organization are arrested and imprisoned. Each prisoner is in solitary confinement with no means of communicating with the other. The prosecutors lack sufficient evidence to convict the pair on the principal charge, but they have enough to convict both on a lesser charge. Simultaneously, the prosecutors offer each prisoner a bargain. Each prisoner is given the opportunity either to betray the other by testifying that the other committed the crime, or to cooperate with the other by remaining silent. The possible outcomes are:
>
> - If A and B each betray the other, each of them serves two years in prison
> - If A betrays B but B remains silent, A will be set free and B will serve three years in prison
> - If A remains silent but B betrays A, A will serve three years in prison and B will be set free
> - If A and B both remain silent, both of them will serve only one year in prison (on the lesser charge)
>
> A representation of the situation in the form of a pay-off matrix has been shown below.

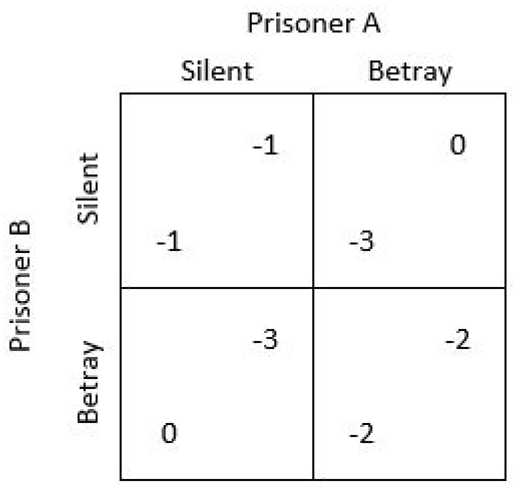

Prisoner A
Silent
Betray
Prisoner B
Silent
Betray
-1
-1
0
-3
-3
0
-2
-2

Figure 4: Pay-Off Matrix

16.1 Strictly dominated strategies

A naive answer to the prisoner dilemma would be that prisoner A and B should both stay silent. After all, it is the choice which causes the least ”total” harm to both of them and it is the co-operative choice. However, prisoner A and B are both rational decision-makers. Let us observe this a little closely. If prisoner A knows that prisoner B will stay silent, then staying silent will cost them a year in prison, but betraying B will get them no years at all. In this situation, betraying B is the choice with a better pay-off. What if A knew that B is going to betray them? Staying silent will mean 3 years of prison while betraying will result in only 2 years. In this situation, as well, betraying is the better choice. In other words, the strategy ”to betray” *strictly dominates* the strategy ”stay silent”. Hence, staying silent is a *strictly dominated strategy* and a rational decision-maker will thus, never choose it.

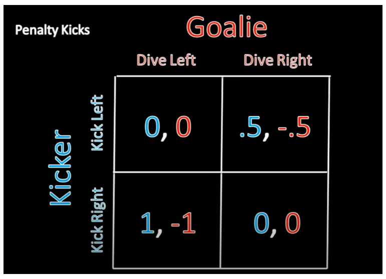
16.2 Pure strategies

A pure strategy provides a complete definition of how a player will play a game. Pure strategy can be thought about as a plan subject to the observations they make during the course of the game of play. In the above example, after observing that staying silent is a strictly dominated strategy, the player forms their strategy - betray. This is different from a mixed strategy. (Pure stategy can be considered a degenrate form of mixed strategy in which the probability for one choice is 1 and the other is 0.)

16.3 Mixed strategies

A mixed strategy is an assignment of a probability to each pure strategy. When enlisting mixed strategy, it is often because the game doesn't allow for a rational description in specifying a pure strategy for the game. This allows for a player to randomly select a pure strategy. Let us consider the following example to understand the concept of a mixed strategy.

> In a soccer penalty kick, the kicker must choose whether to kick to the right or left side of the goal, and simultaneously the goalie must decide which way to block it. Also, the kicker has a direction they are best at shooting, which is right if they are left-footed. The matrix for the soccer game illustrates this situation.



Penalty Kicks

		Goalie	
		Dive Left	Dive Right
Kicker	Kick Left	0, 0	.5, -.5
	Kick Right	1, -1	0, 0

Figure 5: Pay-Off Matrix

> It assumes that if the goalie guesses correctly, the kick is blocked, which is set to the base payoff of 0 for both players. If the goalie guesses wrong, the kick is more likely to go in if it is to the right (payoffs of +1 for the kicker and -1 for

the goalie) than if it is to the right (the lower payoff of +0.5 to kicker and -0.5 to goalie). This game has no pure-strategy equilibrium, because one player or the other would deviate from any profile of strategies—for example, (Left, Left) is not an equilibrium because the Kicker would deviate to Right and increase his payoff from 0 to 1. The kicker's mixed-strategy equilibrium is found from the fact that they will deviate from randomizing unless their payoffs from Left Kick and Right Kick are exactly equal. If the goalie dives left with probability g, the kicker's expected payoff from Kick Left is

$$g(0) + (1 - g)(0.5)$$

and from Kick Right is

$$g(1) + (1 - g)(0)$$

Equating these yields

$$g = 2/3$$

Similarly, the goalie is willing to randomize only if the kicker chooses mixed strategy probability k such that dive Left's payoff of

$$k(0) + (1 - k)(-1)$$

equals dive Right's payoff of

$$k(-0.5) + (1 - k)(0)$$

So,

$$k = 1/3$$

Thus, the mixed-strategy equilibrium is (Prob(Kick Left) = 1/3, Prob(dive Left) = 2/3).

Note that in equilibrium, the kicker kicks to their best side only 1/3 of the time. That is because the goalie is guarding that side more. Also note that in equilibrium, the kicker is indifferent which way they kick, but for it to be an equilibrium they must choose exactly 1/3 probability.

16.4 Nash Equilibria

In his famous paper, John Forbes Nash Jr. proved that there is an equilibrium for every finite game. Nash showed that the optimal outcome of a game is one where no player has an incentive to deviate from their chosen strategy after considering an opponent's choice. One can divide Nash equilibria into two types. Pure strategy Nash equilibria are Nash equilibria where all players are playing pure strategies. Mixed strategy Nash equilibria are equilibria where at least one player is playing a mixed strategy. While Nash proved that every finite game has a Nash equilibrium, not all have pure strategy Nash equilibria.

17 Hotelling Game

The Hotelling Game was one of the examples that involved Nash Equilibria and was observed much prior to John Forbes Nash Jr. stating this. This Game was first observed by Harold Hotelling and published as "Stability in Competition" in Economic Journal in 1929.

17.1 Game Situation

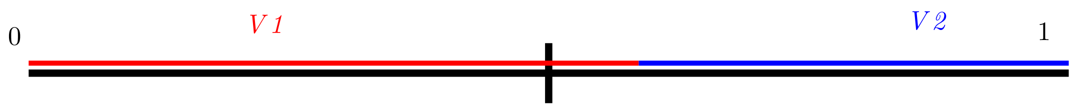


Figure 6: Hotelling's Game

The Simplified Game is as follows:

> There are 2 vendors selling Identical Products at Identical Price in an area which can be represented as a 1D line, consumers being uniformly distributed. Let the consumers prefer the Vendor closest to them. Then the optimum location either of the vendors would choose to maximise their profits is?

17.2 Approach

There are infinite strategies possible for either of the vendors. The best approach would be to constraint down pay-offs of each vendor and narrow down possibilities or even find one of the Nash Equilibria.
Consider the cases when Vendor 1 decides to stay situated at the mid. We can claim that his pay-off is at least half of the total consumers that are present.

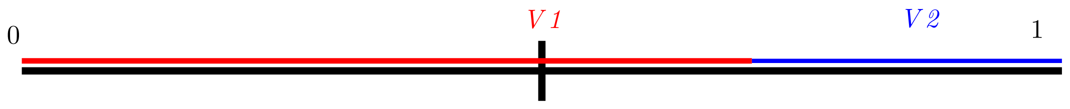


Figure 7: V_1 Optimum Choice

17.3 Nash Equilibrium

V_1 is indifferent to what V_2 chooses as he will always cover at least half of the market. So The Mid is the Optimum position of V_1. By symmetry of the situation, same logic applies to V_2, she being indifferent to V_1 choices if she is at the mid. This gives rise to a Nash Equilibrium, as all players are playing optimally and are indifferent to the other players' choices.

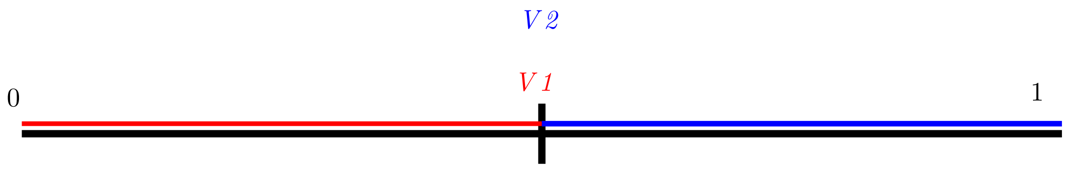


Figure 8: Nash Equilibrium of Hotelling's Game

In this equilibrium, both the vendors are covering half of the market. Are their other location possibilities where this can happen? Of course when both the vendors are equidistant from the Mid. But we can show that such locations are not Nash Equilibria.

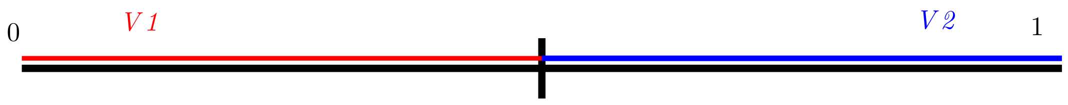


Figure 9: Possible Equilibria? No

Why is the above not an equilibrium? Cause Let's say V_1 decides to move more towards the Mid, so he covers more consumers, increasing his pay-off, thus being better off than before. Same logic applies to V_2, she can do the same. Do remember that while deciding pay-offs, we assume that all other players are playing like they were before. Clearly, both the vendors will tend to move towards the Mid, which was our Nash Equilibrium in the first place.

17.4 Median Voter Theorem in Elections

Now the Relevance of this Game in our project? This Game is a simplified abstraction of quite a wider range of games that take place in real life. For example, Candidates of an Election tend to follow such Nash Equilibrium if the Voting Audience can be divided into such two extrema. The distribution of audience will matter only on the location of the Nash Equilibrium, but the central idea remains the same.

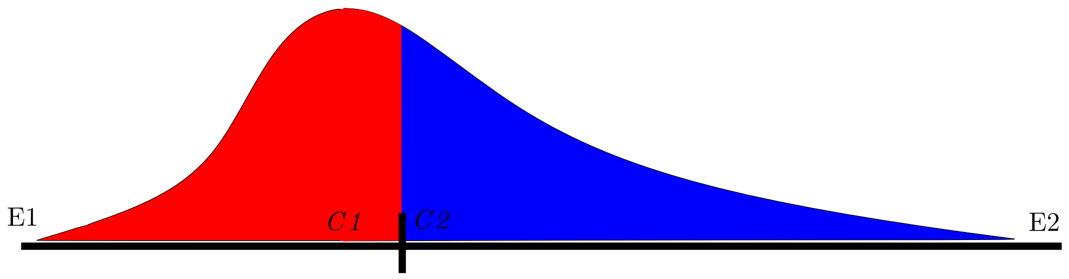


Figure 10: Hotteling Law in Elections

The 2 candidates will tend to attract voters from both extrema, therefore they (their ideas/ appeal/ efforts to attract either of the bodies) will lie on the median of the distribution, i.e. near the beliefs of the median voters. This is the median voter theorem.

18 Second Price Auctions

Another interesting topic discussed during the project. The situation is as follows:

> An Auction is being held such that no bidder knows what is the price other bidders are offering. There can be any number of bidders. The Auctioneer then checks are bids, and declares the highest bid as the winner of the auction. The twist is, the winner has to pay the price equivalent to the second highest bid. What is one of the most likely situations to occur in such an auction?

This again is an infinite strategy game with infinite players. This, for obvious reasons, can't be approached through grid-and-check method. But We can observe something interesting nevertheless.
Notice the scenario when all bidders bid the price exactly how they value the item. We claim that this scene is a Nash Equilibrium.
Reminding the definition of the Nash Equilibrium, no player has the incentive to deviate from his current strategy. Incentives are decided by pay-off, which in turn is decided by the amount paid and the value the player gives to the item in context. We can prove this situation to be a nash equilibrium by showing that a winning and a losing bidder, neither of them have any incentive to change their bid.

- Winning Bidder - If he tries to increase his bid, he still has to pay the second highest bid price. Else if he tries to decrease his bid value
  - greater than the second highest bid, and he still has to pay the latter amount
  - less than the second highest bid, and he now gains nothing, decreasing his pay-off.
- Losing Bidder - If she decreases the bid price, she gains nothing like she was initially. Else if she increases her bid,
  - less than the winning bid, she still gains nothing.
  - greater than the winning bid, she becomes the winner, but now has to pay more for what she values the item, decreasing the pay-off.

Therefore no bidder has the incentive to change his/her strategy, making the scenario a Nash Equilibrium.
What's interesting about this particular Nash Equilibrium, is that this equilibrium is created with incomplete knowledge about the game. No player knows what others are doing, but still the result can converge to this particular situation. This comes under incomplete-information Game Theory, which we now scratched the surface.

19 Voting and Auction Mechanisms based on Game Theory

Below is a list of common electoral,auction and school choice mechanisms based on game theory. Those implemented in the form as a smart contract using Solidity are marked.

Electoral Mechanisms	
First Past the Post	✓
Two Round System	✓
Block System	✓
General Ticket	✓
Cumulative Voting	✓
Single Non-Transferable Vote	✓
Single Transferable Vote	✓
Instant Run-Off	✓
Mixed Member Proportional Representation	✓
Borda Count	✓
Binomial System	✓
Parallel Voting	✓
Plurality-at-large	✗
Party-list Proportional Representation	✗
Majority bonus system	✗

Auction Mechanisms	
English Auctions	✓
Dutch Auctions	✓
Japanese Auctions	✓
First Price Sealed Bid	✓
Vickrey Auctions	✓
School Choice Mechanisms	
Boston Student Assignment	✓
Columbus Student Assignment	✓
Top Trading Cycles	✓
Gale-Shapley Student Optimal Stable	✗

19.1 Voting Mechanisms

Following are the various voting mechanisms that have been implemented as a smart contract using solidity.

1. **First past the post** It is the simplest and most common electoral mechanism across the world. It is formally also known as single-member plurality voting. Voters cast their vote for a candidate of their choice, and the candidate who receives the most votes wins (even if the top candidate gets less than 50 percent of the votes).

2. **Two Round System** One way to avoid candidates being elected with only a small proportion of the popular vote is to hold a second ballot if no one candidate wins a majority on the first round. In the Two-Round System, voters cast a single vote for their preferred candidate. The election proceeds to a second round only if in the first round no candidate has received at least some other prescribed percentage. In the second round, those candidates who received above a prescribed proportion of the votes, would be candidates in the second round. Any remaining candidate is free to withdraw from the second round. This system is generally used by having the prescribed percentage as 50%(simple majority) and just the top 2 candidates for the second round to yield a single winner.

3. **Block voting system** The Block Vote is simply the use of First Past the Post (FPTP) (see First Past the Post (FPTP)) voting in multi-member districts. Each elector is given as many votes as there are seats to be filled, and they are usually free to vote for individual candidates regardless of party affiliation. In most Block Vote systems they may use as many, or as few, votes as they wish.

4. **General Ticket** General ticket representation is a type of block voting in which voters opt for a party, or a team's set list of candidates, and the highest-polling one becomes the winner. It, unless tempered to apply to a specific proportion, arrives at a 100% return for one party's list who become representatives for the membership or representative positions which are the purpose of the election.

5. **Cumulative Voting** This method allows voters to cast all of their votes for a single candidate as a representative when multiple representatives are elected from a single electoral district. In contrast, in "regular" or "statutory" voting, voters may not give more than one vote to any single candidate.

6. **Single Non-transferable Vote** Single non-transferable vote or SNTV is an electoral system used in multi-member districts. It is a generalization of first-past-the-post, applied to multi-member districts. Unlike Block Voting, where each voter casts multiple votes, under SNTV each voter casts just one vote. In any election, each voter casts one vote for one candidate in a multi-candidate race for multiple offices. Posts are filled by the candidates with the most votes.

7. **Single Transferable Vote** STV uses multi-member districts, with voters ranking candidates in order of preference on the ballot paper. In most cases the voters are not required to rank-order all candidates; if they wish they can mark only one. After the total number of first-preference votes are counted, the count then begins by establishing the quota of votes required for the election of a single candidate. The first stage of the count is to ascertain the total number of first-preference votes for each candidate. Any candidate who has more first preferences than the quota is immediately elected. If no-one has achieved the quota, the candidate with the lowest number of first preferences is eliminated, with his or her second preferences being redistributed to the candidates left in the race.

8. **Instant Run-Off Voting** IRV is a proportional system, technically single transferable vote (STV) electing one candidate. In instant-runoff voting, each voter ranks the list of candidates in order of preference. The voter marks a '1' for the most preferred candidate, a '2' for the second-most preferred, and so forth, in ascending order. When counting,the candidate appearing as the first preference on the fewest ballots is eliminated. These elimination rounds are continued till one candidate achieves a majority.

9. **Multiple Member Proportional Representation** Mixed-member proportional representation (MMP or MMPR) is a mixed electoral system in which voters get two votes: one to decide the representative for their single-seat constituency, and one for a political party. Seats in the legislature are filled first by the successful constituency candidates, and second, by party candidates based on the percentage of nationwide or region-wide votes that each party received. The constituency representatives are elected using first-past-the-post voting (FPTP) or another plurality/majoritarian system. The nationwide or region-wide party representatives are, in most jurisdictions, drawn from published party lists, similar to party-list proportional representation.

10. **Borda Count** The Borda count is a family of positional voting rules which gives each candidate, for each ballot, a number of points corresponding to the number of candidates ranked lower. The lowest-ranked candidate gets 0 points, the next-lowest gets 1 point,

etc., and the highest-ranked candidate gets $(n-1)$ points, where n is the number of candidates. Once all votes have been counted, the option or candidate with the most points is the winner. The Borda count is intended to elect broadly acceptable options or candidates, rather than those preferred by a majority, and so is often described as a consensus-based voting system rather than a majoritarian one.

11. **Binomial System** The system works in the following manner: Parties and independent candidates group themselves into lists or coalitions. Each list proposes up to two candidates per electoral region, province, or other geographical unit. Votes are first tallied by list instead of by candidate, and unless the list which obtained the first majority has double the voting as the second majority, each of the two lists gets one of their candidates, the one who got the most voting, into office. In other words, the binomial system basically means that the first and the second majority get equal representation unless the first majority doubles the second.

12. **Parallel Voting** Parallel voting describes a mixed electoral system where voters in effect participate in two separate elections for a single chamber using different systems, and where the results in one election have little or no impact on the results of the other. Under a supplementary member system (SM), which is a form of semi-proportional representation, a portion of seats in the legislature are filled by pluralities in single-member constituencies. The remainder are filled from party lists, with parties often needing to have polled a certain quota, typically a small percentage, in order to achieve representation, as is common in many proportional systems. Any supplementary seats won by a party are filled from an ordered list of nominated candidates

19.2 Auction Mechanisms

Following are the various auction mechanisms that have been implemented as a smart contract using solidity.

1. **English Auctions** This is the most common and traditional type of auctions that we see. The auctioneer opens the auction by announcing a suggested opening bid, a starting price or reserve for the item on sale. Then, the auctioneer accepts increasingly higher bids from the floor, consisting of buyers with an interest in the item. The highest bidder at any given moment is considered to have the standing bid, which can only be displaced by a higher bid from a competing buyer. If no competing bidder challenges the standing bid within a given time frame, the standing bid becomes the winner, and the item is sold to the highest bidder at a price equal to their bid.

2. **Dutch Auctions** A Dutch auction initially offers an item at a price in excess of the amount the seller expects to receive. The price lowers in steps until a bidder accepts the current price. That bidder wins the auction and pays that price for the item. Dutch auctions are a competitive alternative to a traditional auction, in which customers make bids of increasing value until nobody is willing to bid higher.

3. **Japanese Auctions** An initial price is displayed (usually low to start with). All buyers interested in buying the item at the given price enter the auction. This displayed price increases continuously, or by small discrete steps. Uninterested buyers keep exiting the

auction till only one buyer is left, who is declared the winner and pays the displayed price.

4. **First Price Sealed Bid Auctions** First Price Sealed Bid Auctions are also known as Blind Auctions. In this auction, all bidders simultaneously submit sealed bids so that no bidder knows the bid of any other participant. The highest bidder pays the price that was submitted. This is also a very common type of auction.

5. **Vickrey Auctions** Vickrey Auctions are also a type of sealed bid/blind auctions. Bidders submit written bids without knowing the bid of the other people in the auction. The highest bidder wins but the price paid is the second-highest bid. This arrangement encourages bidders to bid the true value they are willing to pay.

19.3 School Choice Mechanisms

Following are the various school choice mechanisms that have been implemented as a smart contract using solidity.

1. **Boston Student Assignment** In this mechanism, each student submits a preference ranking of the schools. . For each school a priority ordering is determined according to the following hierarchy: First priority for sibling and walk zone, second priority for sibling, third priority for walk zone and then for other students. Students in the same priority group are ordered based on a previously announced lottery. Then the allotment of students is done via multiple rounds, where the kth choice of the remaining students are considered in the kth Round. They are then alloted seats till the school doesn't have a vacancy or no student has the school as the kth choice.

2. **Columbus Student Assignment** This mechanism makes use of a waiting list for each school. Each student may apply to up to three different schools. For some schools, seats are guaranteed for students who live in the school's regular assignment area and the priority among remaining applicants is determined by a random lottery. For the remaining schools, the priority among all applicants is determined by a random lottery. For each school, available seats are offered to students with the highest priority by a lottery office and the remaining applications are put on a waiting list. After receiving an offer a student has three days to accept or decline an offer. If he/she accepts an offer, he/she is assigned a seat; he/she then must decline offers from other schools and he/she is removed from the waiting list of other schools to which he/she has applied. As soon as seats become available at schools because of declined offers, the lottery office makes offers to students on the waiting lists.

3. **Top Trading Cycles** Assign a counter for each school which keeps track of how many seats are still available at the school. Initially set the counters equal to the capacities of the schools. Each student points to her favorite school under her announced preferences. Each school points to the student who has the highest priority for the school. Since the number of students and schools are finite, there is at least one cycle. Moreover, each school can be part of at most one cycle. Similarly, each student can be part of at most one cycle. Every student in a cycle is assigned a seat at the school she points to and is removed. The counter of each school in a cycle is reduced by one and if it reduces to zero, the school is also removed. Counters of all other schools stay put. The cycles

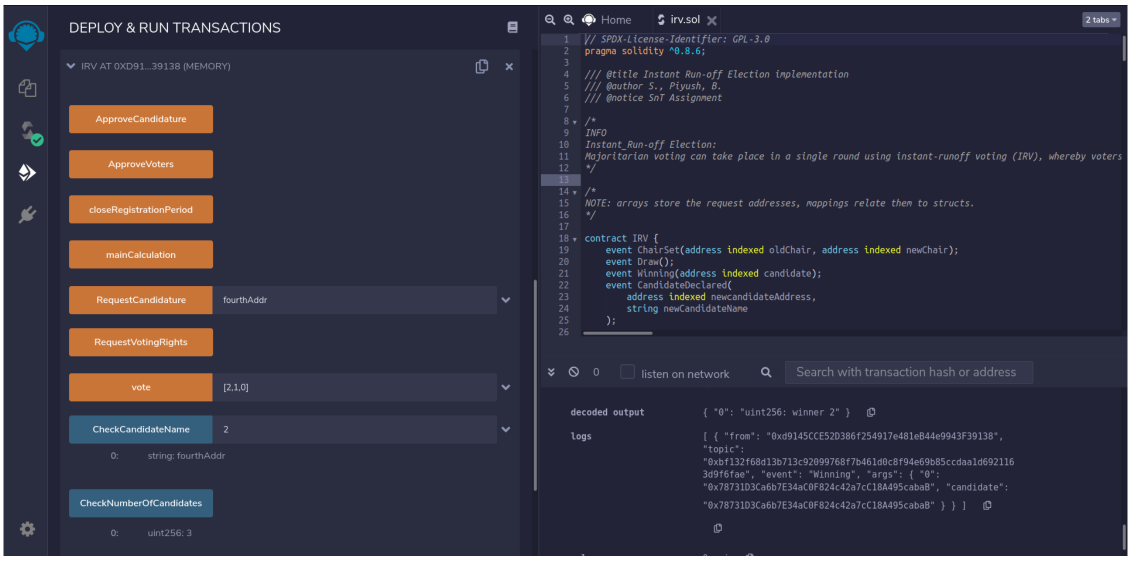
are repeated among the remaining students and remaining schools till each student is assigned a seat.

20 Assignment : Implementing Instant Run-off Voting

This is a type of ranked preferential voting-cum-counting method used in single-seat elections with more than two candidates. Voters pass their support to the candidates, on the basis of preferences, in simpler terms, they have to rank candidates on their ballot.

Some problems with the Instant Run Off voting mechanism,

- IRV is not meant for proportional representation
  - → This implies it is still a winner-takes-all type of election implementation.
  - → It leaves out political minority.
- IRV slightly reduces but does not eliminate most of the enormous numbers of wasted votes in plurality elections.
- It also does not produce multiparty legislatures that truly reflect the variety of political views in the electorate
- IRV eliminates none of the problems associated with redistricting, such as uncompetitive districts and partisan gerrymandering



```
RequestCandidature  fourthAddr
vote  [2,1,0]
CheckCandidateName  2
    0:  string: fourthAddr
CheckNumberOfCandidates
    0:  uint256: 3
```

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.6;

/// @title Instant Run-off Election implementation
/// @author S., Piyush, B.
/// @notice SnT Assignment

/*
INFO
Instant_Run-off Election:
Majoritarian voting can take place in a single round using instant-runoff voting (IRV), whereby voters
*/

/*
NOTE: arrays store the request addresses, mappings relate them to structs.
*/

contract IRV {
    event ChairSet(address indexed oldChair, address indexed newChair);
    event Draw();
    event Winning(address indexed candidate);
    event CandidateDeclared(
        address indexed newcandidateAddress,
        string newCandidateName
    );

```

```
decoded output      { "0": "uint256: winner 2" }
logs                [ { "from": "0xd9145CCE52D386f254917e481eB44e9943F39138",
                    "topic":
                    "0xbf132f68d13b713c92099768f7b461d0c8f94e69b85ccdaa1d692116
                    3d9f6fae", "event": "Winning", "args": { "0":
                    "0x78731D3Ca6b7E34aC0F824c42a7cC18A495cabaB", "candidate":
                    "0x78731D3Ca6b7E34aC0F824c42a7cC18A495cabaB" } } ]
```

20.1 The Registration Period

In this period, people on different addresses can request their voting rights and apply for candidature. The chairperson can then approve their requests. The functions involved in this period are as follows.

1) `RequestVotingRights() public`

Any user who is not already registered as a voter can use this function to request voting rights from the chairperson. This function adds the address of the user to the '`VoterRequests[]`' array.

2) `ApproveVoters() public isChairperson`

   This function can only be called by the chairperson, and approves all the pending requests for voter rights. It then adds the address to the '`voterAddr[]`' array and initializes the '`Voter`' elements linked to the addresses through the '`voters[]`' mapping.

3) `RequestCandidature(string memory name) public`

   Any registered voter can use this function to apply for candidature. Every applicant must input a '`name`' as a string. This will be displayed to other voters if their request is approved. This function adds the user's address to the '`CandidateRequests[]`' array, and stores the '`name`' in a mapping.

4) `ApproveCandidature() public isChairperson`

   This function also, can only be called by the chairperson, and approves all the pending candidature requests. It then adds the addresses to the '`candidateAddr[]`' array and initializes the '`Candidate`' elements linked to the addresses through the '`candidates[]`' mapping. Also, an event '`CandidateDeclared(...)`' is emitted for every approved candidate containing the relevant details.

5) `closeRegistrationPeriod() public isChairperson`

   The chairperson can call this function to officially declare the end of the registration period. After this, calling any of the above functions will result in a revert call.

20.2 The Voting Period

In this period, all registered voters can see the details of contesting candidates and vote. There is no option available for the voters to see the current status of votes of the candidates since this is meaningless for this type of voting mechanism. The functions involved in this period are as follows.

1) `CheckNumberOfCandidates()` and `CheckCandidateName(uint256 k)`

   These functions can be called by any user. The '`CheckNumberOfCandidates()`' returns the total number of registered candidates and '`CheckCandidateName(k)`' can be used to check the name of the `k`'th candidate.

2) `vote(uint256[] memory preferenceList) public`

   Any registered voter can use this function to vote. A valid vote should contain the indexes of all the registered candidates in an array-format with no repetitions. For example, in an election with 5 candidates, $[1, 2, 3, 0, 4]$ would be a valid vote whereas $[1, 2, 3, 2, 0, 4]$ or $[1, 2, 4]$ would be invalid votes.

20.3 The Calculation Period

In this period, all the calculation required to determine the winning candidate is done. The functions involved in this period are as follows.

1) `majorityOrEliminatedCandidate() private returns (uint256 winLoser)`

   This is a helper function returns the the index of the candidate who has achieved majority (> 50% votes), if there is any. In case of no majority, it returns the index of the candidate with the least number of votes, i.e. the candidate which has to be eliminated in the current iteration. In the last iteration, where there are only two candidates competing, it also checks for a draw.

2) `eliminateCandidate(uint256 indexOfLosingCandidate) private`

   This is also a helper function, which eliminates the losing candidate from the preference lists (votes) of all the voters, and then recalculates the number of votes for each candidate according to the updated preference lists.

3) `mainCalculation() public isChairperson returns (uint256 winner)`

   This function integrates the above two helper functions to finally determine the winner of the election process, and emits the event '`Winning(...)`' which contains all the relevant details. In case of a draw, the event '`Draw()`' is emitted. Only the chairperson can call this function and, calling this function also ends the *voting period* and this election process.

20.4 Flaws and Scope of Improvement

1) One of drawbacks of our election implementation, is when multiple candidates tie for the bottom place. A valid Tiebreaker is to eliminate both the candidates, and transform their voters' ballots by deleting these candidates' mentions. The current implementation only eliminates one of the candidates.

   We can fix this by returning an array of the candidates who need to be eliminated, in the '`majorityOrEliminatedCandidate()`' function. Then we can modify the function '`eliminateCandididate(k)`' such that it eliminates all instances of these candidates in all the voters' preference lists.

2) Another drawback is while filling in the candidate preference order, the voter can input multiple instances of the same candidate, which is not valid. We haven't implemented verification to check if all inputs are unique.

   This can either be trusted upon the voter, or otherwise we will have to set more constraints in the input function or the interface with which the voter is interacting with this contract.

3) The functions '`ApproveCandidature()`' and '`ApproveVoters()`' automatically approves the pending requests without any manual approval of the chairperson. For having the option of manual approval for all/some of the requests, further modification is required.

4) By default, 50% or above votes are counted as majority in this contract. For any other value for the majority, one will have to manually change the condition in the contract.

5) In case of a draw, the contract emits Draw log, but fails to return their addresses. One can still do so, by checking all candidates' votes individually.

6) The mechanism of Instant Run Off voting, or at least the above implementation is computationally expensive and hence, is not very gas efficient. The calculation process requires the preferences of each voter to be modified during every elimination. In a practical scenario, the number of voters is going to be very large and looping over this over and over again is going to be very inefficient.

21 Assignment : Implementing Dutch Auctions

A dutch auction is a market structure, primarily based on the idea of initial public offering. The price of something offered is determined after taking in all bids to arrive at the highest price at which the total offering can be sold. In this type of auction, investors place a bid for the amount they are willing to buy in terms of quantity and price.

- One of the main distinguishing features of this mechanism, is the price of item is kept quite high initially, and price is lowered until it gets a bid. Therefore there is actually no competition between bidders during the auction itself.
- The price with the highest number of bidders is selected as the offering price, so entire amount is sold at a single price. This price may not be the highest nor the lowest.
- Such auction is quite popular for day-to-day used items or consumables.

21.1 Basic Details about the Contract

1) `constructor(max_initial_price, min_base_price, number_of_shares)`

   While deploying the contract, one must input the starting price and the minimum price of an individual share, along with the total number of shares that are up for auctioning. The constructor sets the address which deploys the contract, as the *Auctioneer*. It initializes the relevant variables and also marks the starting time of the auction.

2) List of modifiers used,

   i) `isAuctioneer` : Requires the sender to be the auctioneer, otherwise a revert is called.

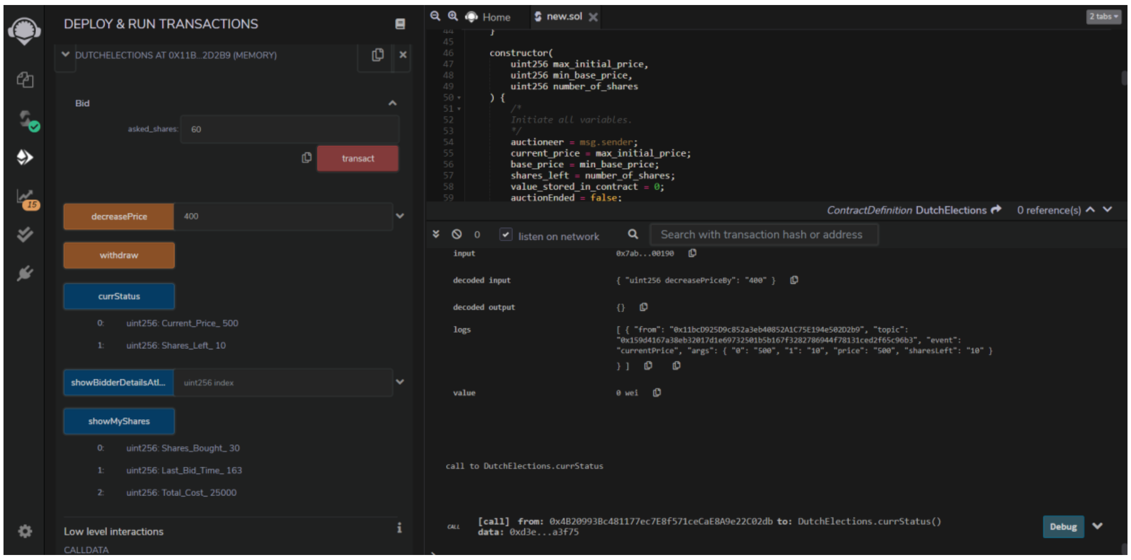
   ii) `auctionGoingOn` : Requires the auction to be going on currently, otherwise a revert is called. This is used to prevent bids made after the auction process concludes.

3) List of events created,

   i) `currentPrice(uint256 price, uint256 sharesLeft)`

   ii) `bidderWhoWon(address bidder, uint256 numberOfShares)`

   iii) `AuctionEnded(uint256 shares_left)`



```
asked_shares: 60
decreasePrice 400
currStatus
0:  uint256: Current_Price_ 500
1:  uint256: Shares_Left_ 10
showBidderDetailsAtI...  uint256 index
showMyShares
0:  uint256: Shares_Bought_ 30
1:  uint256: Last_Bid_Time_ 163
2:  uint256: Total_Cost_ 25000
```

```
    }

    constructor(
        uint256 max_initial_price,
        uint256 min_base_price,
        uint256 number_of_shares
    ) {
        /*
        Initiate all variables.
        */
        auctioneer = msg.sender;
        current_price = max_initial_price;
        base_price = min_base_price;
        shares_left = number_of_shares;
        value_stored_in_contract = 0;
        auctionEnded = false;
```

ContractDefinition DutchElections 0 reference(s)

```
input            0x7ab...00190
decoded input    { "uint256 decreasePriceBy": "400" }
decoded output   {}
logs             [ { "from": "0x11bcD925D9c852a3eb40852A1C75E194e502D2b9", "topic":
                 "0x159d4167a38eb32017d1e69732501b5b167f3282786944f78131ced2f65c96b3", "event":
                 "currentPrice", "args": { "0": "500", "1": "10", "price": "500", "sharesLeft": "10" }
                 } ]
value            0 wei

call to DutchElections.currStatus

CALL  [call] from: 0x4B20993Bc481177ec7E8f571ceCaE8A9e22C02db to: DutchElections.currStatus()
      data: 0xd3e...a3f75
```

21.2 Bidding Starts

Following are the functions present in this contract :

1) `decreasePrice(uint decreasePriceBy) public isAuctioneer auctionGoingOn`

   This function is used by the auctioneer to decrease the price of shares throughout the auction process. An event '`currentPrice(...)`' is emmited whenever the price is changed. In case the current price is already equal to the minimum price set initially, the function '`endAuction()`' is automatically called and the auction concludes. Only the auctioneer is allowed to call this function.

2) `currStatus() public view returns(uint Current_Price, uint Shares_Left)`

   Anyone can call this function to get the current status of the auction process. It tells you about the number of shares that are still left for auctioning, and also the current price for an individual share.

3) `Bid(uint asked_shares) public payable auctionGoingOn returns(Warning)`

   This function can be called by any address to bid at the current price. One has to enter the number of shares they want to buy. This function is a payable function and the sender must also send the corresponding amount of wei required to buy the requested number of shares at the current price. This can be done by setting the the *value* before calling the function, and in case excess amount is sent to the contract, the function returns this amount back to the user. The function stores the relevant details about the bid, like the time, price, and number of shares bought and then returns a message informing the user about the status of the bid and the transaction. In case of a successful bid, an event '`bidderWhoWon(...)`' is emitted.

4) `showMyShares() public view returns(Shares_Bought,Last_Bid_Time,Total_Cost)`

   This function lets the user know about their bidding details. It outputs the total number of shares bought by the user until the function is called, the time of their latest bid, and the total amount payed to the contract.

5) `withdraw() public returns(Warning)`

   While using the '`Bid(...)`' function, if the process of returning excess amount payed is somehow disrupted, the user can use this function to request the withdrawal of any pending returns. The function then returns the status of the withdrawal request.

6) `showBidderDetailsAtIndex(index) public view`
   `returns ( Address, Shares_Bought, Total_Cost, Last_Bid_Time )`

   This function can be called by any address, and returns the details of any of the winning bidders. Apart from providing transparency, this function is particularly helpful for the seller, and auctioneer in order to keep a record of the winning bidders.

7) `endAuction() private`

   This is a private function which is used to officially end the auction process and transfer the complete amount received by the contract to the auctioneer's account. It also emits the event '`AuctionEnded(...)`'. This function is automatically called inside the '`decreasePrice(...)`' function when the current price has already reached the minimum price, and in the '`Bid(...)`' function when the number of shares reaches zero.

22 Student Assignment Mechanism

In a School Choice Problem, there are a number of students, each of whom should be assigned a seat at one of a number of schools. Each school has a maximum capacity but there is no shortage of the total seats. Each student has strict preferences over all schools, and each school has a strict priority ordering of all students.

22.1 Boston Student Assignment Mechanism

Let's focus on one of the mechanisms implemented in our project, the Boston-School Student Assignment Mechanism. Some key features are:

- Each student can submit his/her preference sheet, which is submitted with the timestamp of registration.
- Each school has a priority order of accepting students:
  1. Siblings already in School and lives within Walk-zone.
  2. Siblings already in School
  3. Lives within Walk-zone
  4. Others
- Students under same category are preferred according to their time of registration.
- Enrollment is done in rounds:
  1. **Round 1:** All first preferences are considered, and seats are filled in according to the priority order. This is done until either all seats are full, or there are no more students which have that school in the first priority.

2. **Round 2:** All second preferences are considered, and seats are filled in according to the priority order. This is done until either all seats of the school are full, or no more students have that school in the second preference rank.

⋮

K. **Round K:** All K^{th} preference are considered, and seats are filled in according to the priority order. This is done until either all seats of the school are full, or no more students have that school in their K^{th} preference. This is the stage at which either all students are alloted, or all seats of each school is filled in.

22.2 Implementation of this Mechanism

Implementation is here. Summarising the Contract:

- Deploy the contract using Owner's or Authority's address.
- Authority can only register schools. `RegisterSchools` is called by passing school's name and vacancy:
  - `RegisterStudent (Studentname0, 1)` ; from Student0's address
  - `RegisterStudent (Studentname1, -1)` ; from Student1's address
  - `RegisterStudent (Studentname2, -1)` ; from Student2's address

  Each time `RegisterStudent` is called, they are pushed into `StudentList`. Rolln0 is the index of the Student in the `StudentList`. By default each student's `schoolAssigned` is -1.
- After registering, each student must submit preference ranking of all schools.
- Each time `SetPriority(SchoolCode, schooldistance or walkzone)` is called, it is added to respective Students's Priority array at the end, which is inserted in Priority array.
- The order in Students's Priority array represents the preference of the student and each index can be called as `preference_index`. Preference for a school cannot be changed.
- `show_priority_index` can be run by Students to check their priority would be by passing Student's Rolln0 and a `preference_index`.
- `StartAssigningSchools` can only be called by Authority to assign schools to the students. So, this is called only when all students have filled their prefernces for all the schools. Each time a student is assigned a school, he get's added to `ResultList` at the end.
- Each student can access their results by calling `ShowResults` using their address to get their assigned school's name. Authority can access results using `ResultsList` by passing an index to see who has been assigned which school at that point.

23 Team

Mid-Term Eval Contributors

Documentation:	S Pradeep	Ayush Anand	Piyush Beegala	B.Anshuman
Presentation:	Girik Maskara	Somya Gupta	Anubha Tripathi	Dhwanit Balwani

End-Term Eval Contributors

Documentation:	S Pradeep	B.Anshuman	Piyush Beegala	Ayush Anand
Presentation:	Anubha Tripathi	Ayush Anand	Aditya Subramanian	

Mentees

Aadeesh Jain	Abhay Yadav	Aditya Subramanian	Ansh Saxena
Anshit Arya	Anubha Tripathi	Aryaman Singhal	Aryash Pateriya
Atur Gupta	Ayush Anand	B.Anshuman	Dhwanit Balwani
Dipanshu Garg	Dwarq Nikhesh Babu	Girik Maskara	Moin Ahmed
Muditt Khurana	Mukesh Kumar	Patil Aditya Bharat	Piyush Beegala
Pranab Pandey	Prashant Verma	Rishabh Dugaye	Rohit Kumar
S Pradeep	Saket Dhakar	Shorya Kumar	Somya Gupta
Suket Raj	Yash Goel	Piyush Agarwal	

Mentors

Atharv Singh Patlan	Rohan Baijal
Shubhankar Gambhir	Yatharth Goswami