Overall

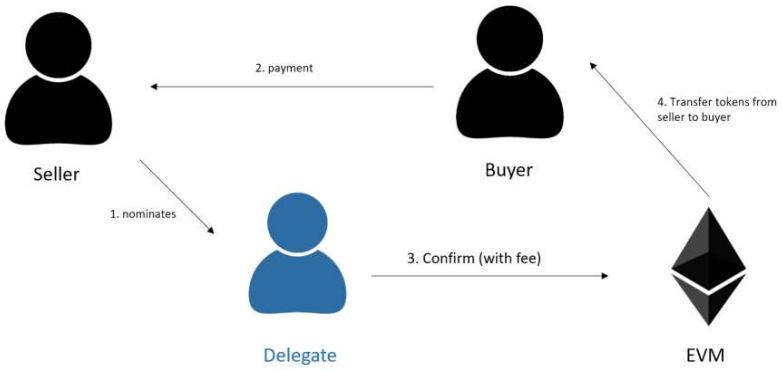
As running programs on a blockchain is expensive, most design choices I have made in this model is to minimize running cost. To the greatest extent possible, I tried to record information using Boolean mappings, which is quick to store and reference. Compared to run-time, storage space is not that valuable.

For-loops is kept at a minimum, unless there is no better choices. Even so, the function with For-loops will be run by people who trigger them, which are students in most cases.

Due to a lack of proven foul-proof time-triggered mechanism on the Ethereum EVM, in my design, the administrators will be responsible to set a deadline for each round, and when the round closes, trigger the roundCloses() function to confirm or refund students. In every function triggered by the university, this is the costliest function to run. Still, the sorting part of bids is already handled at trigger-time, which is beared by students, and the university only pays for updating the records.

After the round closes, Administrators can set the time for a new deadline, and a new round commences.

Trading



Since the university do not wish to have any funds passing through their contracts, a new role called Delegate is introduced. In this design, they are someone to witness a truthful transfer of funds. Any ETH address can be a delegate, but it is supposed to be a trusted third-party for both sides.

To start a sale, the seller nominates a delegate and the number of tokens by startSale() function. After this, the delegate will receive Ether from the buyer. After it is confirmed, the Delegate will trigger the confirmSale() function, along with exactly 10% of value in Ether. This will unlock the number of tokens in the seller's account, and trigger a transfer function to transfer the tokens from seller to the buyer.

As long as the Delegate is trust-worthy, he/she could acquire good will and become a trusted delegate, even demanding fees from trades. In my opinion, this would serve the purpose of educating students about free markets.

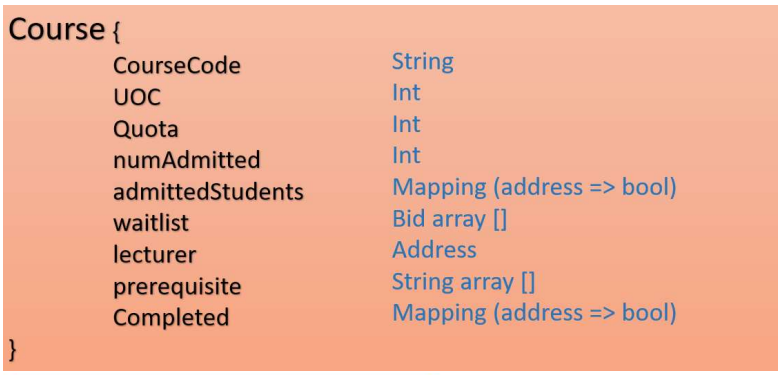
Bidding

There are two functions associated with bidding, bidCourse() and bidCourseWithSignature(). In bidCourse(), the student attaches the bid amount and the course he wish to bid for. Checking is done to make sure the call is valid, and then a findBid() internal function is called to try find a bid made earlier by the student. If it is found, the bid is removed. After this, the bid is pushed into the back of the waitlist, and bubbled up in an O(n) fashion. Bid with the highest price is at index 0, and insertion starts checking from end of the waitlist. This way, the insertion gas fee is proportional to the amount the student is willing to pay. Bidding activity are restricted for time before the deadline, which

Sorting mechanism:

In this design, a simple bubble sort is used to place a new bid in the waiting list; This is O(n) time, which means run time and gas is proportional to size of the list. In terms of algorithm, there are definitely faster choices like QuickSort, which runs in O(logn) time. However, taking into account the re-ordering nature of sorting, most programs can't avoid inserting and deleting from the array, which are all costly operations in solidity. Even if there are performance enhances, they only show up at really large N sizes, which is an overkill for courses with at most 200-300 students. That's the reason I have chosen to stick with bubble sort.

Data structures: Course



The course structure stores the most information in the admission system. Notably, there are 2 arrays, the waitlist and the prerequisite. It is quite hard to avoid using array for waitlist due to the nature of bids; Sorting needs to be done in a efficient and quick manner. Pre-requisites consist of course names, and will be used to check students' eligibility. Luckily, most courses' number of pre-requisites will likely be a small integer, so pre-req checking don't take a lot of time.

Permissions

Permission-locked functions like adding new Admins and students are done using modifiers. For singular roles like COO, the address is stored directly. For roles like Admins and Lecturers, a mapping is used to store their addresses. This ensures that functions will only be called by people it's meant for.

```
modifier onlyCOO() {
    require (msg.sender == COO, "Unauthorized");
    _;
}

modifier onlyAdmin() {
    require (isAdmin(msg.sender), "Unauthorized");
    _;
}

modifier onlyStudent() {
    require (isStudent(msg.sender), "Unauthorized");
    _;
}
```

Lecturer permission to enroll

Lecturer's approval of a student consist of message "CourseCode" + "studentAddress". In my design, I assume the student obtained the signature from the professor in some way. In order to submit the bid, the student submit a bid by triggering the bidCourseWithSign() function, including the hash and the signature signed by the lecturer. In this function, the hash submitted will first be verified by the message scheme above, to ensure the hash is genuine. Next, the hash and the signature will be put into the ecrecover() in-build function and recover the signer. Finally, if the address recovered is indeed the lecturer, the rest of the function is the same as bidCourse(), without checking pre-requisites.

Something to develop in the future is to encode additional parameters, like time-out periods, where the lecturer only grant the approval for a limited amount of time.

Running Cost analysis

Gas analysis from Remix

Action	by?	Execution gas cost	Cost ($2450 per ETH)
Deploy contract	UNSW	5996872	2644.6
roundCloses	UNSW	162604	71.7
newCourse	UNSW	130075	57.4
bidCourse (Bidding into 5 people)	Student	102315	45.1
bidCourse (re-bid in waitlist of 3)	Student	98075	43.3
bidCourse (1 out of 1)	Student	90324	39.8
confirmSale	Delegate	73542	32.4
buyToken	Student	67279	29.7
addPreReq	UNSW	48323	21.3
startSale	Student	47159	20.8
addStudentCompleteRec	UNSW	26299	11.6
newStudent	UNSW	23635	10.4
newAdmin	UNSW	22551	9.9
cashOut	UNSW	9913	4.4
setDeadline	UNSW	6675	2.9
setFee	UNSW	6603	2.9

Gas Price 180 **Gwei**

The above is a table detailing the gas price to run functions in my design, obtained from Remix. The gas price is an average taken recently at non-peak hours on etherscan.io/gastracker.

Understandably, the most pricey function is the deployment of the contract. Since deployment and adding new course are one-time costs, let's look more deeply into the cost to run the system and to maintain a session.

From the table, we see that the closing of each round is costly, since it runs through the waiting list of each course one-by-one and refund students. This number will probably grow proportional with the number of classes and students.

Although it seems expensive, UNSW can minimize this cost by decreasing number of bidding rounds so it won't hurt as much. Otherwise, the more expensive functions are triggered by the students, more so when you are bidding high into a large list of students. All the other admin costs (setFee, setDeadline) are quite cheap compared to those mentioned above. Notably, trigger of NewStudent also increase proportionally with number of students.

Platform suitability

Running a course admission system on a blockchain is definitely a cool idea. It is similar to DAO: It is immutable and automatic, makes decision based on a known set of rules. It also enhances transparency of the admission system. Compare to the traditional centralized platform, it is more resilient to attacks, and cheap to maintain. However, there are also a few drawbacks:

1. Cost to students: Crypto prices fluctuates a lot, Ether has witnessed some substantial price rises and falls. Interacting with such a system requires having Ether on hand. This gives an unfair advantage to rich students, which I am not sure if UNSW wants to promote. Some students may be reluctant to change their bid because of the high triggering cost.
2. Network Congestion: As one of the blockchains to implement the first games on tokens, Ethereum network has seen some congestion at times. No one can guarantee that in the future during course admission period some popular game would came out on Ethereum and drives the gas prices sky-high;
3. Risk: The Ethereum network has experienced quite a number of hacks in the past, some of them resulted in a hard fork of the entire network. When such an event occurs again, even if it has nothing to do with the admission contract, it may bring systematic uncertainty to the school; This is a risk that traditional centralized computer systems don't have;
4. Security: Smart contracts are not foul proof; The more external dependencies it rely on, the more vulnerable it is to attacks. Malicious attackers have shown to find bugs in smart contracts and exploit them to their advantage. Writing smart contracts with best practices usually take more effort, cost more gas to run and time to audit. This is a trade-off that the University may need to take.
5. Privacy: Every transaction is trackable on the blockchain. Although addresses are anonymous, once they are revealed you have everything to show. For example, if you have used your ETH account to play card games, the University Administrator could tell by looking into your account. This shows that with great transparency blockchain also comes with some compromises on personal privacy.

In terms of the contract itself, I have tried my best to make things simple and robust, while strictly refraining from using external calls. In theory, malicious actors shall not be able to take advantage. Still, as EVM codes are running on remote random mining machines, there's always a chance some mining machine throws a pointer error, and the system could exhibit unexpected behaviour.

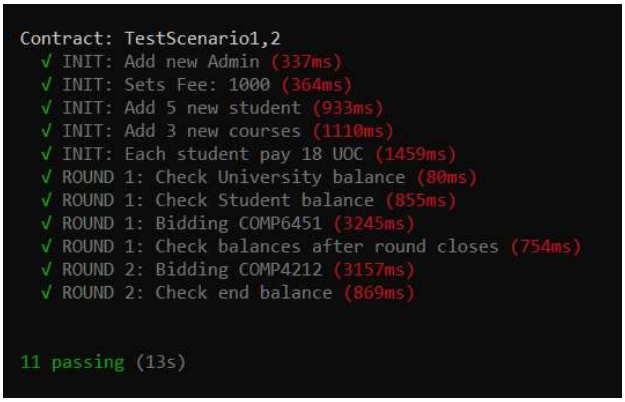
Testing

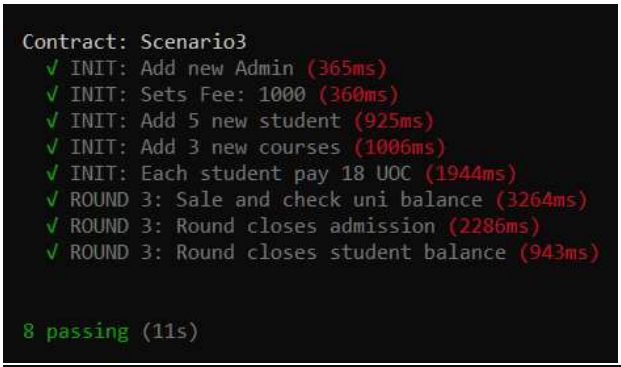
All test scripts are written to run on Truffle javascript, tested with solidity version 0.8.1. Sample scenarios given are executed without failure. Apart from the cases provided, I have also written test scripts for the COO actions and multiple pre-requisite courses. To reproduce my results, in the truffle directory, run the command as follows:

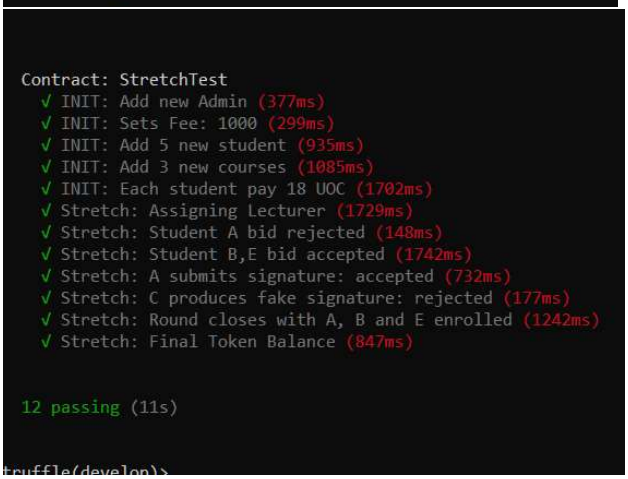
Truffle migrate –reset

Truffle develop

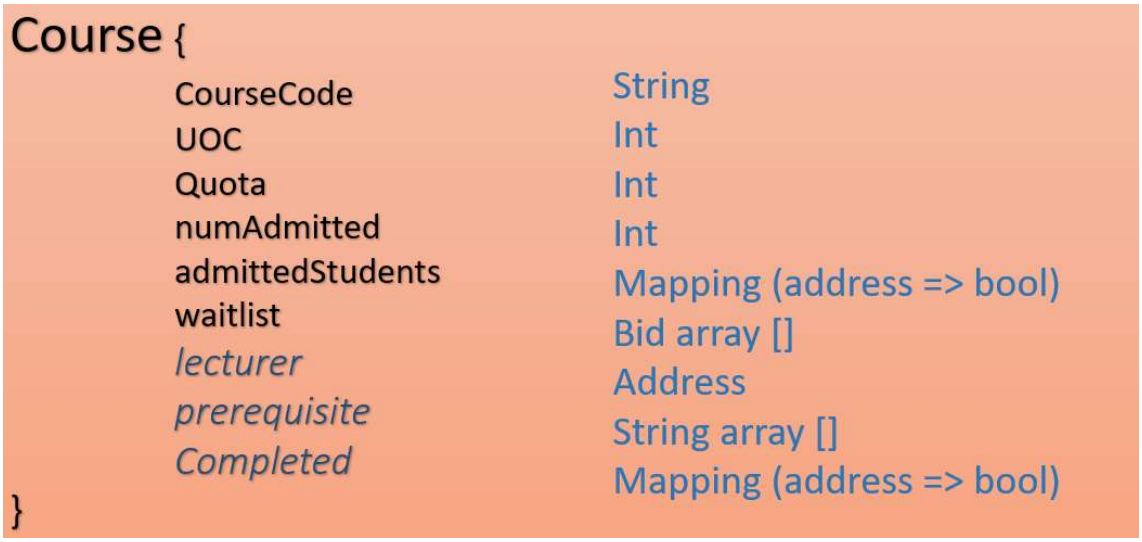
Truffle test







Stretch Part



For the stretch goals , 3 new fields (lecturer, pre-requisite & Completed) are added to the Course structure.

Lecturer is an information that is provided by Administrators by triggering the LecturerCourses() function.

The pre-requisite field is an array that stores pre-requisites courses of the course. Students bidding on the course should have either the lecturer's signature, or completed every course of the prerequisites array.

Finally, the completed field is a mapping that maps addresses that have completed the course.

The Bid course function has been updated to check for pre-requisites: Apart from the usual validity checking, if there are prerequisite courses in the array it will loop through everyone of them before proceeding to inserting the bid.

Signature

As mentioned earlier, the bidCourseWithSign() is the function to bid course with professor's signature. There are 6 parameters:

```
// same as above function, only change the prerequisite requirement with validate signature
function bidCourseWithSign(string memory code, uint amount, bytes32 hash, uint8 _v, bytes32 _r, bytes32 _s)
```

The student has to work out the _v, _r, _s values of the signature, presumably by some web3 front-end tools. In this function, we have to check that the hash is not re-used or obtained from someone. To do this, the hash is reworked by keccak256 hashing the combined string of the course code and his address. After confirming the hash is intact, it is put in the ecrecover function together with v,r,s. This should recover the signature of the address that signed the message.

In the test script for stretch goal, the splitting part is done by javascript string slicing.