CS 8803/4803 Blockchain and Cryptocurrencies.

Solidity Programming Assignment 1

Name: Abhishek Mukesh Sharma

1.1 Part 0, Setting up Truffle (0 Points)

Please see the attached `readme.txt` for instructions on how to set up Truffle. We highly recommend that you go over the CryptoZombies Tutorial.

1.2 Part 1 (80 Points)

You are provided a skeleton contract, and you are to implement the empty functions it contains.

The attached `readme.txt` contains detailed instructions and examples.

We will run 10 test cases, each is worth 8 points (5 test cases for iterative bubble sort, 5 for recursive).

1.3 Part 2 (20 Points)

Answer the following questions, please provide formal and precise explanations.

Each question is worth 2 points but question 5, which is worth 6 points.
Bonus question is worth 2 points.

Solidity

1. What version of the solidity compiler are you using with truffle? How did you determine this using truffle?

   - **Answer:** I am using the 0.8.11 solidity compiler. I determined by the setting in the `truffle-config.js` file in the `compilers` section.

2. Explain the parameters for the first line. `pragma solidity`. What do each of the $\wedge$,<=,< operators do?

   - **Answer:** `pragma solidity` is a keyword that helps specify the solidity compiler version that needs to be used for the solidity file.[1]
     The uses of various operators is defined as follows -
     - $\wedge$ - This operator specifies to choose the latest minor version of the compiler given a major version.

[1] Reference - `https://learning.oreilly.com/library/view/solidity-programming-essentials/9781788831383/581dfde3-516d-42c1-b4b7-bab1e9d1db8d.xhtml`

  - <= - This operator specifies to choose any available compiler version that is either older than or equal to the specified version.
  - < - This operator specifies to choose any available compiler version that is either older than the specified version.

3. What is the maximum size of `uint` type in solidity? Give your answer in terms of bits. Why do you think this size is the default for `uint` in Solidity?

   - **Answer:** `uint` is the alias for `uint256` datatype, which has the maximum size of 256 bits.[2] I think this size may be default in Solidity because this size make `uint` as the perfect container to store 256 bits hash outputs from SHA256 hashing functions, which is used in abundance in smart contracts and Ethereum. **ALso EVM works in 256 word size.**

4. What is the difference between a `pure` and a `view` function in Solidity? Please provide and explain a use case for a `pure` function and a use case for a `view` function.

   - **Answer:** `pure` function in Solidity refers to such a function in contract which does not read, or modify storage space in the contract. `pure` functions can be used for mathematical helper functions, like computing some mathematical operation on a bunch of data and returning the result, while not viewing or modifying state (storage space) of the contract.
     `view` function in Solidity refers to such a function in contract which only reads, but does not modify storage space in the contract. `view` function can be used to give information about the contract state to external parties.

5. How much gas was used to deploy (not execute) the 1. iterative bubble sort function and the 2. recursive bubble sort function of your contract? Explain how you were able to compute this. Compare the gas used between each function and give a reason which would explain the difference.

   - **Answer:** While deploying a contract, we don't execute any functions, so the gas used is dependent only on the compiled EVM bytecode (of the functions and the contract), and a deployment cost (which is generally independent of the contents of the contract). Hence, to find out the gas used while deploying the individual function, I simply removed the source code of the other function in the contract, and compiled and deployed the contract. The gas used to deploy -
     (a) iterative bubble sort function is 370433 gas.
     (b) recursive bubble sort function is 390089 gas.
     (c) both the functions is 495285 gas.
     (d) none of the functions (just the contract skeleton without the function skeleton) is 67066 gas.

     These gas calculation was done using truffle deployment on a Ganache local blockchain network.
     The simple reason of the difference between the gas used by each function is that the gas used calculation is dependent on the EVM bytecode size or the amount of opcodes as part of the compiled function. Each function is compiled into a EVM bytecode, which

[2]Reference - `https://docs.soliditylang.org/en/v0.8.11/types.html`

is an amalgamation of several different type of opcodes (kind of similar to assembly language, like `PUSH`, `MOV`, etc.) Since both the functions (iterative and recursive) have different implementation, the resulting compiled binary is of different size. For the iterative function, the deployed bytecode is of 2950 bytes, and for the recursive function, the deployed bytecode is of 3132 bytes. Because deploying involves actually storing the contract on the blockchain (which requires gas), the gas used is dependent on how much storage or contract size is being deployed. Hence the difference.

6. Is the gas used to deploy deterministic? Explain.

   - **Answer:** To deploy a contract, the Ethereum transaction just contains the compiled bytecode of the contract in the data field and the receipient is just set to 0. The gas used for this transaction simply depends on the size of the compiled bytecode. Hence, the gas used to deploy is deterministic.

7. How many local variables can you have in a Solidity function?

   - **Answer:** Because of the stack size limit in EVM (where the local variables in a Solidity function are store), you can have no more than 16 local variables in a Solidity function.

8. What is `msg.sender`?

   - **Answer:** `msg.sender` is the caller or the sender of a transaction in the Ethereum blockchain. `msg.sender` can hold an address of a user's account or an address of a contract (contract calling another contract) for the transaction.

(Bonus) Is Solidity Turing Complete? Please provide an in depth answer. There are a lot of wrong answers for both yes and no online, so be careful what you read.

- **Answer:** I believe that Solidity is a Turing Complete programming language and here are a few reasons why I think that's the case -
  - The first is that Solidity is based on OR can simulate a replicated state machine. We know that state machines by themselves are Turing Complete because they can replicate the working of a Turing machine. Solidity can implement a function such that $SxI \rightarrow SxO$, where S is a set of states of the system, I is the possible input set and O is the possible output set. Hence Solidity, as an independent language, is definitely a Turing Complete programming language.
  - Many people argue that Solidity is not a Turing Complete language because it does not allow infinite memory as part of its execution. But we must understand that this memory restriction in Solidity is not because of the programming language, rather the environment in which Ethereum Virtual Machine (EVM) is set. The EVM is a global distributed network and it has to enforce multiple restrictions on the memory usage because the memory is finite in the world, and huge amounts of memory can make the Ethereum blockchain slow down (because much of the computational resources will be stuck handling the huge amount of memory). Hence, Solidity as a programming language does not have these memory restrictions. It is because we use Solidity to program contracts for EVM, these restrictions have been applied on the language to help user's develop a code that is compatible to run on EVM.

  Because of the above reasons, I believe that Solidity is a Turing Complete language.

Your deliverables include only two files. `Sorter_YOUR_NAME.sol` which is the skeleton contract provided, but fully implemented, and `Report_YOUR_NAME.pdf`, which is your solution to the questions in part 2. Please do not zip the files.