How Not To Run Out of Gas in Ethereum

Or, how to code scalable smart contracts.

Alberto Cuesta Cañada

Follow

Coinmonks

--

2

Listen

Share

Man's respect for the imponderables varies, yet there is always a limit. — H. P. Lovecraft

I've been designing and coding Ethereum blockchain solutions for a while. If you have been following the space you'll have noticed that we do things in a different way around here. We obsess about minimalist code, put up with ridiculous limitations and we always act like our messes will appear in the news.

In this article I'm going to talk about one of those limitations which constrain Ethereum developers to the core: The block size. Along with immutability this is easily the biggest constraint to blockchain development.

Unlike in a normal computer the Ethereum network needs to be computed in blocks, and each block can run a limited amount of code. The exact limit varies, but currently it is about 10 million gas. Each Ethereum EVM operation has a different gas cost, but at the end you need to remember that reading one data element from storage is 200 gas, and writing one data element to storage is 20000 gas.

If you run the maths you will find that a single block can have about 50,000 read operations, 500 write operations, or a combination of the two.

When people talk about blockchain being a solution for automating business they don't always get the concept right. Blockchain automates reactions according to immutable order sets.

If I deploy a contract to the Ethereum mainnet that says that to anyone who sends me a cryptokitty, I will mint and send a cryptodog, that will happen forever. Action is followed by deterministic reaction.

What the Ethereum blockchain doesn't do is to run a smart contract algorithm that doesn't end.

Try to do that and you will receive the dreaded out of gas

error. For each action my smart contract can do a reaction costing no more than 10 million gas. That means that we need to use our computing resources sparingly or break tasks down into steps. If I want to distribute dividends to shareholders using a smart contract, each one will have to come and ask for the dividend.

If I run a loop to distribute dividends I will run out of gas before getting to the 500th shareholder.

I like it when coding everyday applications leads me to maths and basic data structures. It's like going back to university. Happy days.

What are the limits?

When coding a smart contract you need to be very careful with loops. A loop is an invitation to an out of gas

error ruining your application.

But to code completely without loops is not fun either.

The key to code smart contracts with the maximum utility is to be very careful with the data structures that we use, to know the limits to computation inherent to block gas limits, and to break down work into separate calls when everything else fails.

To talk about computational limits we will need to use a bit of O notation. If you need a refresher go to the wikipedia and read about O(1), O(log N) and O(N), we won't need any others for now. I'll give you a clue, all Ethereum smart contracts need to run in the tiny Excellent sliver below:

If we consider the gas costs of 200 for a read operation and 20000 for a write operation, and a block gas limit of 10 million gas, then we can make some assertions about the algorithms that we can run in a block.

Linear Algorithms

Basic data structures depend on algorithms that tend to be O(N). This for example would be to keep data in an array and loop through it to do things.

Any O(N) algorithm reading on a data structure will run out of gas if the data structure grows to about N = 50000. For simplicity you can assume that this includes just one write operation as well. We can talk about this as a search-and-write operation.

Any O(N) algorithm writing on all elements of a data structure will run out of gas if the data structure grows to about N = 500. I'll call this a multiple-write operation.

This makes more sense when explained in a real world context. If you have a token that keeps track of all token holders, and for some operation you need to check all of them before updating one contract variable then you can't have more than 50,000 token holders. If your token gives rewards to token holders, and you update all balances in the same call, then your maximum number of token holders is about 500.

Logarithmic Algorithms

There are more complex data structures where the algorithms that manipulate data are O(log N). That means that to find a specific value in a data structure containing N elements you only have to take log N steps which is a much smaller number.

Any O(log2 N) algorithm reading on a data structure will run out of gas if the data structure grows to about N = 2**50000. For simplicity you can assume that this includes just one write operation as well. We can talk about this as a search-andThat means that if your algorithm to search in a data structure is O(log2 N), your smart contract will scale.-write operation. The maximum number that can be represented in Solidity is 2**256 and that is also the maximum number of elements that you can hold in any Solidity data structure.

That means that if your algorithm to search in a data structure is O(log2 N), your smart contract will scale.

Any O(log2 N) algorithm writing on a data structure wouldn't write on all N elements of the data structure, at maximum it would do it on log2 N of them. This means that a O(log2 N) algorithm with multiple writes would run out of gas if the data structure grows to N = 2**500 which is still larger than the maximum number that exists in Solidity.

That means that if your algorithm to write

in a data structure is O(log2 N), your smart contract will scale.

Which algorithm should I use?

I like to make things easy for myself, and for others. Now that we know the general limits and their reasons we can go back to university and map out everything that we can and cannot do:

There are basically four data structures in computer science:

1. Arrays.
2. Linked Lists.
3. Hash Tables.
4. Trees.

Hash Table

Hash tables are quite an advanced data structure in most computing languages, except in Solidity. Everything in Solidity is a Hash Table, which we call mapping

. Even arrays are implemented as mappings under the hood. When I implement Linked Lists I use mappings. When mappings is all you got, mappings is all you use.

Reading from a mapping is always O(1), writing to a mapping is always O(1). There is no search capability built-in, for that you need to implement one of the other data structures. All this means that you should use just mappings whenever you can, and you'll be safe. The size limit for a mapping in Solidity is 2**256.

Arrays

Arrays are a funny contraption in Solidity, but they are also the only built-in data structure that can hold multiple elements and supports a search function.

Arrays can hold 2**256 elements if you don’t need to go through all positions in a single call, and you only insert or remove elements at the end. If your need to check all positions in an array before you write to storage you will need to limit your array to a length of about 50 thousand, possibly less. Arrays shouldn’t be used if you have to insert anywhere except at the end.

Linked Lists

Linked Lists are your data structure of choice when you need to preserve the insertion order, and also when you want to insert in arbitrary positions. You can use them to hold 2**256 elements like arrays for access and arbitrary insert. Also the same as arrays, if you need to visit all positions in a single call you need to limit their length to 50 thousand elements.

You can use Linked Lists to keep your data elements in a specific order by forcing the inserts to happen in the appropriate position. This insert will have a cost of O(N) reads and O(1) write, so it will limit the length to your list to a few tens of thousands without further refinement.

Trees

Trees are the data structure that you use in Solidity if you need to efficiently search in an ordered data set. They are complex but there are a couple of implementations (Order Statistics Tree, Red Black Tree) out there that you can use if you feel brave. All operations in a tree have a complexity of O(log N) which means that you can maintain a tree of astronomical size.

If you use a tree to store data, then you have no practical limits to structure size for search-and-write operations, and a limit of a thousand million million elements for multiple-write operations. Still you will never do more than a few hundred write operations in a single call.

However, using trees comes with its own downsides. They are complex to write and test. They are costly to deploy. In my opinion using such a complex data structure is a huge risk to your project.

Do you have any proof of all this?

Don’t trust me, test by yourself. You can use this small contract to test how many read or write operations fit in a block:

There are some extra gas costs to manage those loops and to make the read operation a state changing transaction by emitting an event, but I would expect these functions to run out of gas after a few hundreds of writes or tens of thousands of reads. These are the results if I set the block size to 10 million:

Boom.

Conclusion

It took me a while to finally understand the computational limits to smart contracts. In this article I gave a clear and concise guide of what you can and can’t do in a call to an Ethereum smart contract.

1. Everything is a mapping, use them always if you don’t have to search through the contents.
2. Use an array if you need to search and can accept inserting and removing only at the end of it.
3. Use linked lists if you need to search, and insert in arbitrary positions, or keep your data ordered.
4. Use a tree if you need to efficiently search in large data sets.

For anything else, you will need to break up your code into different calls and build a state machine. You might as well consider if you should be coding this in blockchain at all.

If you have to visit all elements of a data structure, your size limit is a few tens of thousands. If you have to write for each element in a data structure then your size limit will be a few hundreds.

Don’t run out of gas!