**Important: This document contains useful information to optimize your Gas Cost in Ethereum Blockchain. The following document contains 90% of the gas optimization techniques in Ethereum**

* We will see which method is useful and which method is useless. (or event harmful)
* We will see how much gas you can save by using each method.
* We will rank all the methods by using a tier list. (A = excellent, D= useless/harmful)

# **1. Deployment optimization vs Call optimization.**

At first, you need to know that there are 2 ways to optimize gas on a Decentralized Application..

## **1. Optimizing Deployment costs**

* You can optimize the cost to deploy a smart contract.
* The cost = 21000 gas (for creating transaction) + 32000 gas (for creating a contract) + CONSTRUCTOR EXECUTION + BYTES DEPLOYED \* 78.125
* So for deploying a smart contract, you will spend at least 53000 gas.
* To minimize gas costs you need to minimize the size of a smart contract and the cost of the constructor.

## **2. Optimize every function call.**

* You can also optimize the cost of calling every function in your smart contract.
* The cost is equal to 21000 gas (for creating transaction) + the function execution cost.
* So the objective will be to reduce the function execution cost.

Note that these 2 means of optimization are not the same!!

A lot of times, you may deploy a larger solidity code (so with higher deployment gas cost) but more optimized for every function call. (So with a smaller calling gas cost)

If you need to choose which type of optimization, you need to use.

I strongly advise you to optimize every function call as the deployment is made only one time and the user experience will be enhanced. (unless your smart contract size exceeds the limit of 24Kb)

# **▶ 2. Ranking Gas optimization methods**

We will rank all the gas optimization methods by using a tier list. (Because there is a difference between saving 2 gas on a transaction of 21000 gas and saving 10000 gas.)

Tier A: If you don’t know these rules, you’re not a TRUE solidity developer.

Tier B: Essential, should be known by all the developers.

Tier C: may be useful sometime, should be known by developers too.

Tied D: Useless/harmful, don’t use them (unless you know what you are doing). You may lose, either your time or the security of the smart contract.

# **▶ 3. Tier A : indispensable**

## **★ A1 : Knowing how much each EVM opcodes cost.**

Here is the 6 of the most costly gas opcodes (among the most used is smart contracts):

* CREATE/CREATE2 -> Deploys a smart contract. (32000 gas)
* SSTORE -> Stores a value in storage. (20000 gas if the slot hasn’t been accessed, 2900 otherwise.)
* EXTCODESIZE -> Get the size in bytes of a smart contract code. (2600 if not accessed before, 100 gas otherwise)
* BALANCE (address(this).balance), same as EXTCODESIZE.
* SLOAD -> Access a value in storage. (2100 gas if not accessed before 100 gas otherwise)
* LOG4 -> Create an event with 4 topics. (1875 gas)

(This is in fact an estimation. Some of them may vary depending on the situation.)

The best website which describes how much each opcode is <https://www.evm.codes/>, it’s very well explained.

If you know that, you’ll better understand solidity and you’ll get a “better” intuition of how much each line of code costs you.

## **★ A2 Use view the modifier**

function getBalance() external view {

return balance;

}

You can easily cut your gas costs to 0 when you call the smart contract if you don’t write to the storage. (moreover, you don’t need to wait 5–20 seconds the response)

## **★ A3 Understand the solidity programming language**

More generally, you need to understand how solidity works, this will allow you to not rely on online tutorials for saving you gas!

Fortunately, compared to others languages:

* Rust
* C++
* Javascript

Solidity is VERY easy, there is a very limited number of things to learn.

For example it’s possible to learn 100% of the solidity language in about 1–2 years but even in 100 years you won’t fully master C++.

## **★ A4 Become good at computer science/algorithms.**

You need to know how different algorithms work, how they can be optimized and what is the “complexity”.

For example let’s say, you want to sort an array. (which arises a lot of times)

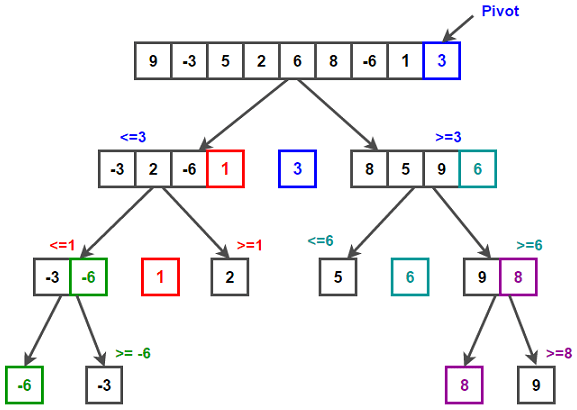
[4,5,108,3,7,1,94,15,99,34,0,24,5,4]

How do you program that?

There are dozens of different sorting algorithms:

Some of them are more efficient than others in some use cases…

You need to choose the best one depending on the situation and it’s not that easy. For example “quickSort” is usually the fastest way to sort an array with a complexity of O(nlog(n))



# **▶ 4. Tier B : Useful**

These tips and tricks can save you a good amount of gas and should be implemented as much as possible.

IMPORTANT NOTE BEFORE STARTING:

* As the not\_optimized() function is situated before optimized() function in the byte code.
* Calling optimized() costs more than not\_optimized() with the same code (22 more gas), so I will subtract 22 gas on each call to optimized() function.

## **★ B1 Batching operations (Saved gas: 21000 gas \* batched transactions)**

Submitting a transaction alone on the blockchain costs a lot of gas (21000 gas to be precise), so if you can find a way to batch transactions for your users, it can save you a good amount of gas.

## **★ B2 change orders of storage slot (20000 gas saved at deployment)**

Ethereum storage is composed of slots of 32 bytes, the problem is that writing is expensive. (Up to 20000 gas using “cold” writing)

Let’s say you have 3 variables:

| uint128 a; //Slot 0 (16 bytes) uint256 b; //Slot 1 (32 bytes) uint128 c; //Slot 2 (16 bytes)  //They uses 3 different storage slots uint256 b; //Slot 0 (32 bytes) uint128 a; //Slot 1 (16 bytes) uint128 c; //Slot 1 (16 bytes) |
| --- |

But if we align the 3 slots well, we can economize 1 slot. (a and c variable will be in the same slot)

Therefore there are 1 less used slot at the deployment. (So 20 000 gas saved)

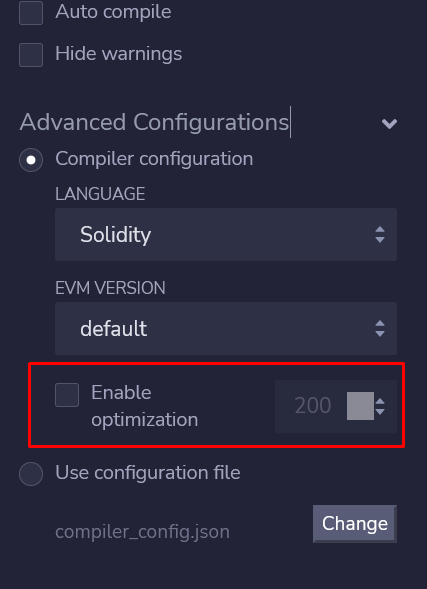
Moreover if you want to access the c variable, but the b variable is already accessed. It will count as a warm access (accessing a storage slot already accessed) so it will cost you 1000 instead of 2900 gas which is pretty significant.

If you don’t understand the storage, the documentation may help you: <https://docs.soliditylang.org/en/v0.8.13/internals/layout_in_storage.html>

## **★ B3 Use the optimizer (Saved gas : 10–50% on deployment & call)**

You can use the solidity built-in optimizer

This is a very simple way to save a lot of gas without learning any new concepts. You need to check the “enable optimization” checkbox.

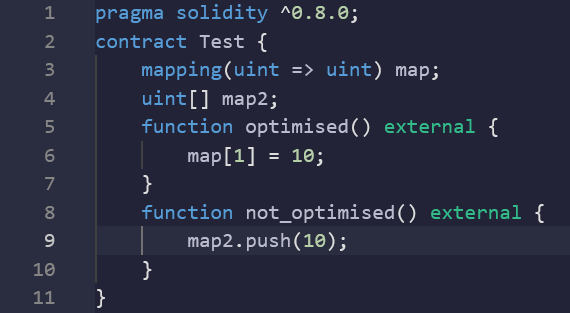


A value near to 1 will optimize gas cost, but a value near to the max (2³²-1) will optimize calls to the function.

In most of the cases you can use the default value (200)

## **★ B4 use mapping instead of array (5000 gas saved per value)**

Mappings are usually less expensive than arrays, but you can’t iterate over them.



not\_optimised() function gas usage: 48409 gas

optimised() function gas usage: 43400 gas

## **★ B5: Use require instead of assert**

Assert should NOT be used by other means than testing purposes. (because when the assertion fails, gas is NOT refunded contrary to require)

## **★ B6: use self destruct (save up to 24000 gas)**

selfdestruct() function destroys the smart contract and refunds 24000 gas.

In a more complex transaction like deploying another smart contract, you can call selfdestruct() on the smart contract to economize some gas.

## **★ B7: Make fewer external calls (amount saved: variable)**

Make a call to another contract only when you’re obligated to do so.

## **★ B8: Look for dead code (saves a variable amount of gas on deployment)**

Sometimes, developers forget to remove useless code like:

require(a == 0)

if (a == 0) {

return true;

} else {

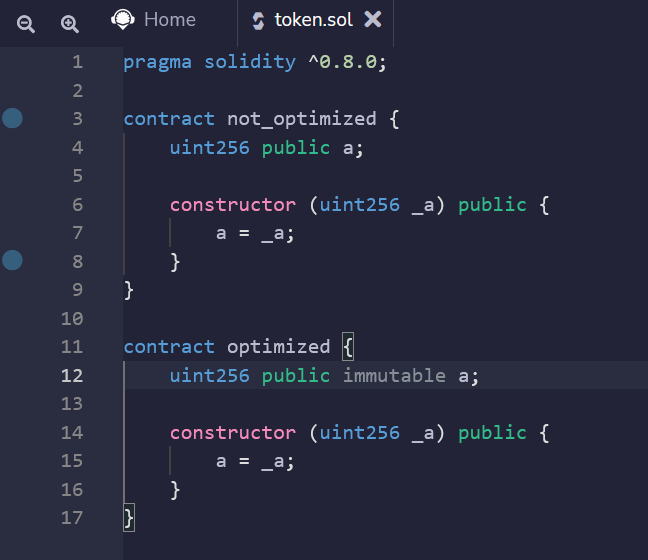
return false

}

This is a dump example because it shows you that some devs can forget to remove useless parts.

## **★ B9 use immutable variables (saves 15000 gas on deployment)**

Use immutable storage variables whenever it’s possible.



not\_optimised contract gas usage: 116800 gas

optimised contract gas usage: 101013 gas

## **★ B10: storing data in events (up to 20 000 gas saved on function call)**

If you don’t need to access the data onchain in solidity, you can store it using events.

| pragma solidity ^0.8.0; contract Test { address store;  event Store(uint256 indexed key,address indexed data);   function optimised() external {  emit Store(1,0xaaC5322e456d45E7b6c452038836C5631C2AeBc0);  } function not\_optimised() external {  store = 0xaaC5322e456d45E7b6c452038836C5631C2AeBc0;  } } |
| --- |

not\_optimised() function gas usage: 43353 gas

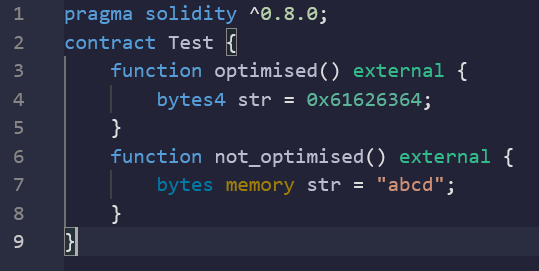
optimised() function gas usage: 22747 gas

# **▶ 5. Tier C : Why not but will not change your life as a Blockchain Developer**

These tips can save you a fair amount of gas and cost you nothing.

## **★ C1 Use static size type in solidity (about 50 gas saved)**

Static size types (like bool, uint256, bytes5) are cheaper than dynamic size type (like string or bytes)

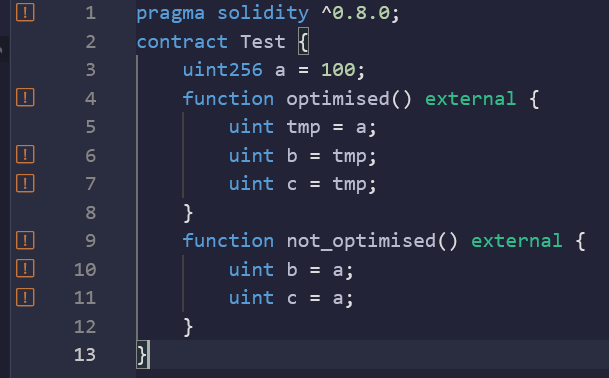


not\_optimised() function gas usage: 21255 gas

optimised() function gas usage: 21227 gas

## **★ C2 Cold access vs warm access (70 saved gas)**

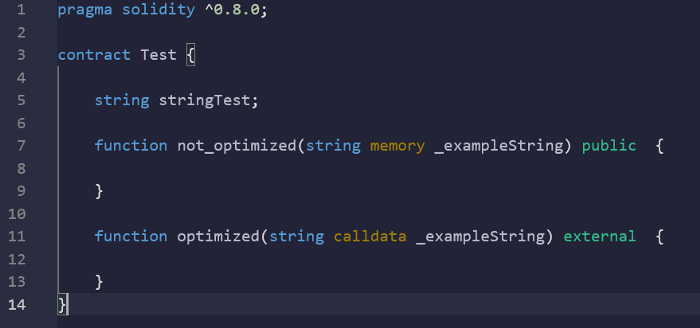
Don’t access the same storage variable twice.



not\_optimised() function gas usage: 23412 gas

optimised() function gas usage: 23347 gas

## **★ C3: using calldata instead of memory (450 saved gas per call)**



not\_optimised() function gas usage: 22442 gas

optimised() function gas usage: 21994 gas

## **★ C4: Use Indexed events (62 gas saved on function call per topic)**

You can mark each event as indexed as bellow:

Indexed events allow events to be searched more easily.

| contract Test {  event Testa(address a,address b);  event Testa2(address indexed a,address indexed b);    function not\_optimised() external {  emit Testa(0x..., 0x...);  }  function optimised() external {  emit Testa2(0... ,0...);  } } |
| --- |

Here the function optimised() uses 135 less gas than not\_optimised().

## **★ C5: Changing the order of calls (20–2000 saved gas per call)**

As said in the important note:

We can call every smart contract, you supply in msg.data the signature of the function. (the first 4 bytes of the function keccak256 hash).

Firstly, the EVM will do a “switch” of this signature, to see which function to execute.

switch(msg.data[0:4]) { // compare to signature

case 0x01234567: go to functionA;

case 0x11111111: go to functionB; // cost 22 more gas..

case 0x4913aaaa: go to functionC; // cost 22+22 more gas...

...

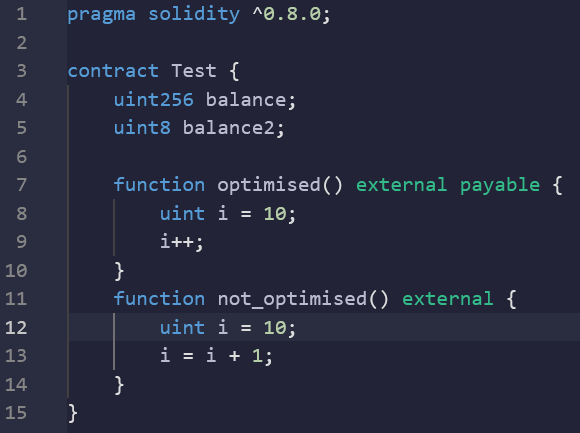
}

As comparing to signature computation use a bit of gas (22 gas), you need to place the most called function in the first place of the switch() by assuring that the signature is in the first place (by changing its name)

You can learn more about this par is my reversing EVM articles: <https://trustchain.medium.com/reversing-and-debugging-evm-smart-contracts-392fdadef32d>

**★ C6 use i++ instead of i = i+1 (62 gas saved on each call)**

This sounds like a joke, but it seems to work.

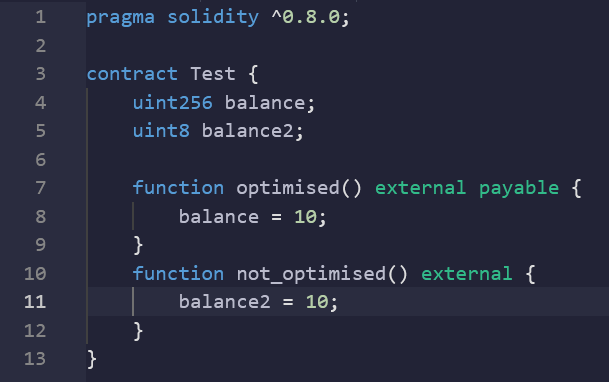


not\_optimised() function gas usage: 21401 gas

optimised() function gas usage: 21339 gas

## **★ C7: Use uint256 instead of uintXX on storage (55 gas saved per call)**

To write/access data on storage, the EVM uses 32 bytes (= 256 bits slots), by using smaller types than uint256, the EVM needs to perform additional operations to assure the conversion.So better to use uint256 instead of uint8 for storing data.

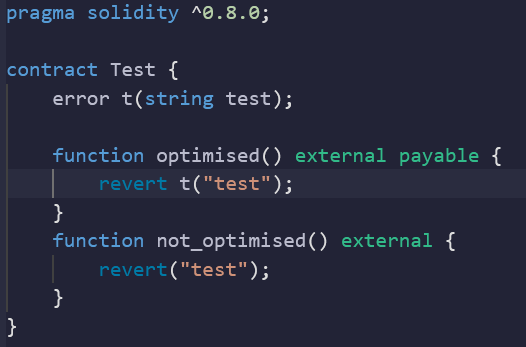


not\_optimised() function gas usage: 43353gas

optimised() function gas usage: 43298 gas

**★ C8: Create a custom error (24 gas saved per call)**

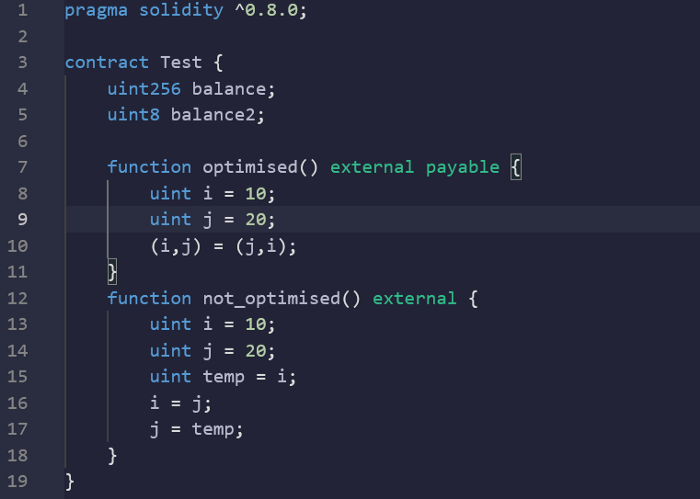
You can create a custom error on solidity, and revert with it like below.



not\_optimised() function gas usage: 21476 gas

optimised() function gas usage: 21474 gas

## **★ C9: Exchanging 2 variables by using a tuple (a, b) = (b, a) (saves 5 gas on every call)**



We put it on the C tier because the notation is way more readable.

not\_optimised() function gas usage: 21241 gas

optimised() function gas usage: 21236 gas

# **▶ 6. Tier D : Useless/harmful**

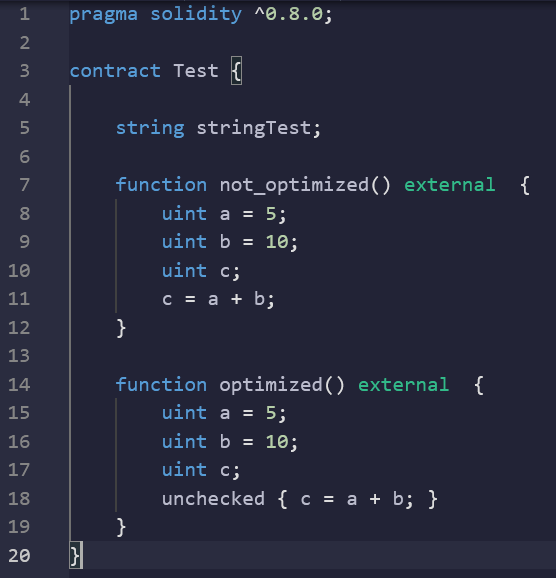
There optimizing gas tips are useless, don’t lose your time for saving a low amount of gas it will be shadowed anyway by the 21000 gas transaction.

## **★ D1: using unchecked (gas saved: 150/operation)**

Since solidity 0.8.0, operations like + \* / — are checked every time if there is an overflow/underflow.

But it cost a little bit of gas to verify if there is an overflow/underflow

Unless you are doing a lot of operations in a single function call (like in for loops), you should NOT use unchecked.



not\_optimised() function gas usage: 21419 gas

optimised() function gas usage: 21253 gas

## **★ D2 changing and optimizing the byte-code yourself**

Don’t do this unless you have a good reason to do so, some of the functions may end up with undefined behavior and security issues.

Use the optimizer instead or implement a very strict testing policy.

## **★ D3 Delete Errors string messages (about 78.125 gas saved on deployment cost per characters)**

Don’t do it, the code may be harder to debug for you and for users.

For example, someone can replace:

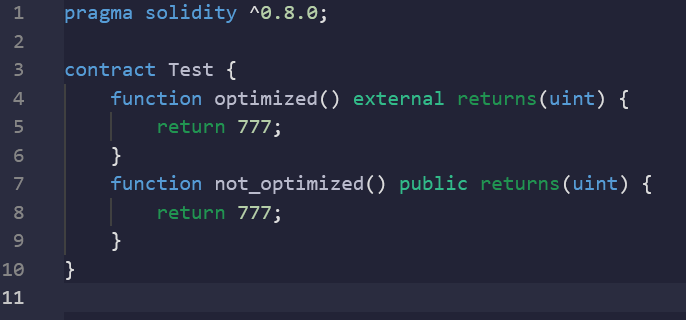
| require(account != address(0), "ERC20: mint to the zero address"); by: require(account != address(0)); |
| --- |

## **★ D4 using low level assembly (gas saved: variable)**

If you don't know what you’re doing, DON’T do solidity assembly. You will end up having a lot more chances to have a security hole on your contract for a marginal saved gas amount.

## **★ D5 use external instead of public**

Surprisingly they are not any difference between external and public functions



not\_optimised() function gas usage: 21401 gas

optimised() function gas usage: 21379 gas (22)

## **★ D6 Use left shift instead of multiplication >> (150 saved gas)**

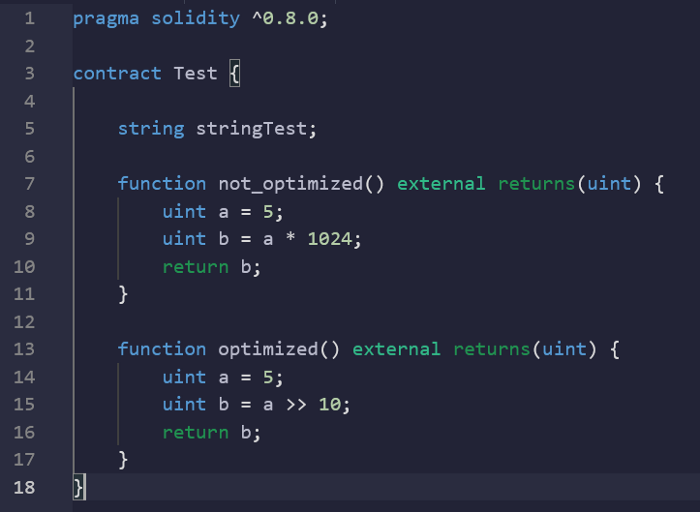
Shifting binary data n times to the left results in multiplying the data by 2^n. Here is an example:

00000001 = 1 dec

00000010 = 2 dec

00000100 = 4 dec

00001000 = 8 dec



not\_optimised() function gas usage: 21615 gas

optimised() function gas usage: 21436 gas

Gains are not bad, but the >> operator doesn't check for an overflow.

## **★ D7: Delete metadata hash (3000–5000 gas on deployment)**

On every smart contract byte code a “hash” is appended in the end.

This is the hash of all the metadata of the smart contract. (Including abi, comments, code…)

As the metadata hash is 32 bytes in length, it can save you a bunch of gas.

But this is quite hard to do, because you have to do it by hand.

So it may be dangerous if executed badly.

## **★ D8 Add payable to every function (24 gas saved per call)**

Adding payable to a function, removes the verification on msg.value. Therefore some gas is saved.

| pragma solidity ^0.8.0; contract Test { address store;  function optimised() external payable {  } function not\_optimised() external {  } } |
| --- |

not\_optimised() function gas usage: 21186 gas

optimised() function gas usage: 21184 gas

It’s not harmful, but not very useful either.

## **★ D9 doing the important work offchain.**

Be careful about the work you’re doing off-chain (Like in JS)

For example, you can’t do security verification off-chain in the website front end, this is very dangerous as anyone may be able to change the JS code in the browser and send bad input to the smart contract.

ALWAYS DO THE SECURITY VERIFICATION ON-CHAIN.