PROBLEM 196

A great mystery of math

Liviu-Ionut Gheorghe
Radu-Tudor Vrinceanu

Context Setup

- A palindromic number is a number that remains the same when its digits are reversed.

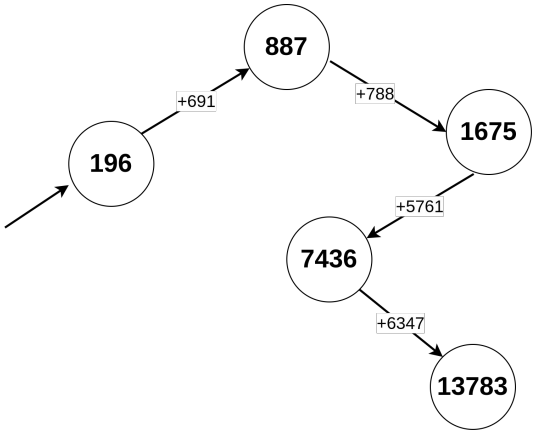
  For example, 12321 and 5665 are palindromic numbers.

- The “Reverse-And-Add” process represents the addition of a number with the number formed by reversing its digits.

  For example, the “Reverse-And-Add” process applied on 67 will yield 67 + 76 = 143.

Problem Overview

- Problem 196 (also known as 196-algorithm) is an unsolved mathematical problem.
- It’s uncertain whether the “Reverse-And-Add” process applied repeatedly, starting from number 196, will ever yield a palindrome.



Problem Overview

- The numbers that cannot form a palindrome through the iterative “Reverse-And-Add” process are called Lychrel numbers.
- At this moment, no base 10 Lychrel number has been proved to exist.
- However, many numbers (including 196) are suspected on heuristic / empirical grounds.
- The name "Lychrel" was given by Wade Van Landingham, who has a very comprehensive page with resources related to this problem (https://www.p196.org/)

Problem Overview

- Because no mathematical proof of the existence of Lychrel numbers has been found so far, people interested in this problem have used computer power over time to make as many iterations of the algorithm as possible.
- This empirical method was adopted in the hope that after a sufficiently large number of iterations, a palindrome will be reached.
- The preferred number of researchers is 196, because it is the smallest number suspected to be Lychrel.

A little bit of history

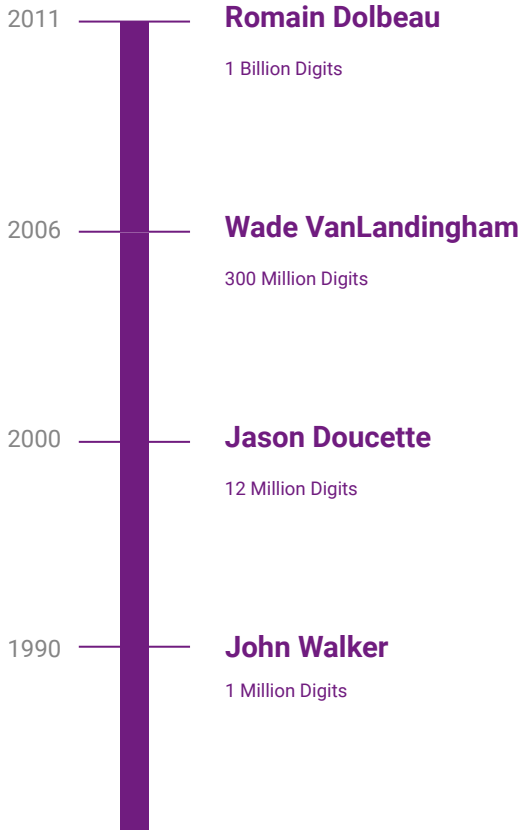
- Over time, many people have been interested in this problem and have worked to make progress in finding a concrete answer.
- The first person to achieve remarkable computational results was John Walker (1987).
- John Walker began his 196 Palindrome Quest on a **Sun 3/260** workstation.
- He wrote a C program to perform the reversal and addition iterations and to check for a palindrome after each step.
- It ran **for almost three years**, then terminated (as instructed) on 24 May 1990 with the message:

```
Stop point reached on pass 2,415,836.

Number contains 1,000,000 digits.
```

A little bit of history

- It took years of computational effort to achieve these results.
- Some of the used algorithms are parallel, running on multiple systems at the same time.
- However, the data confirms that hardware has improved exponentially in the last decades, allowing for much faster calculations.



Our goal

- Our goal is to make our own implementation of the “Reverse-And-Add” algorithm.
- In this way, we can better understand the problem itself and find new ways to optimize it.
- This problem has no direct practical application, but certain optimization techniques used here may be useful for some real-life problems.
- Although there are many efficient implementations, we decided that we want to come up with something original, not just to reproduce something already existing, even if our implementation won’t be more efficient.

Our approach

Naive implementation

We start from the most naive approach for the algorithm and compare its running time with the optimized version

Sequential Algorithm

Decent version of the algorithm that uses the power of a single computer.

Case study for a distributed algorithm

We design a distributed version of the algorithm, which does not exploit the power of a single computer, but of an entire computer suite at the same time.

Testing

We check how fast our sequential algorithm is compared to other implementations - for example the Rust implementation of our instructor)

Naive Implementation

- At each step, create another variable that will hold the digits of the current number in reverse order
- Add this number to the original number
- Check if the result number is palindrome
- Repeat the process until the current number is palindrome

```
import sys


def reverseNumber(number):

   reversed = 0
   while (number > 0):
       reversed = reversed * 10 + number % 10
       number = number //  10

   return reversed


n = int(input("Input the number of iterations:" ))
currentNumber =  196

for iterator in range(n):
   reversed = reverseNumber(currentNumber)

   if reversed == currentNumber:
       print("The number reached a palindrome after  {}
iterations".format(iterator))
       sys.exit(0)

   currentNumber = currentNumber +  reversed
```

Naive Implementation

- As you may guess, this implementation is extremely inefficient. From a certain number of iterations onwards, probably tens or hundreds of thousands, even a modern system would need weeks to calculate the result.
- Because Python is an interpreted language and thus much slower, we can use a compiled one, like Go, Rust or C / C++.
- However, without making additional optimizations this will not help us much.
- We waste too much memory and too much processing power to invert the number each time and add it to the current number.

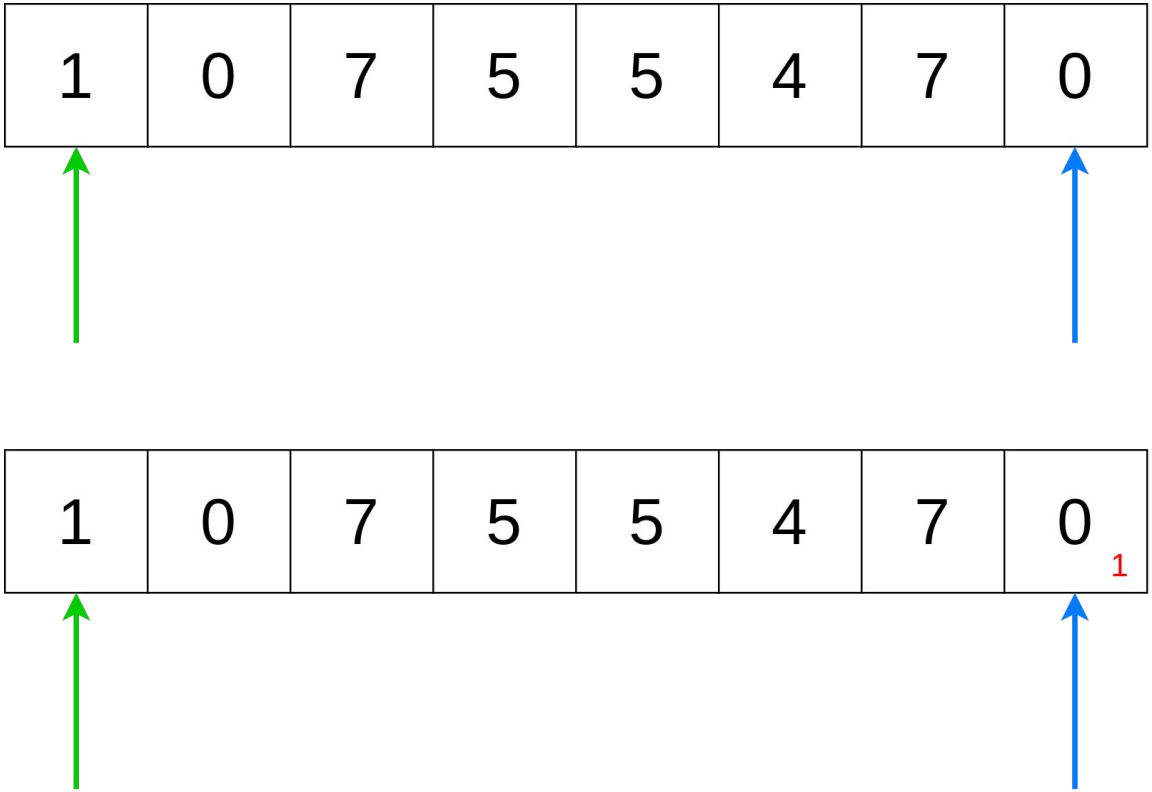
Naive Implementation vs Sequential Algorithm

	Sequential C++	Naive Impl Go	Naive Impl Python
10k iterations	T	~600T	~700T
20k iterations	T	~4500T	~5120T
30k iterations	T	> 10k T	> 10k T

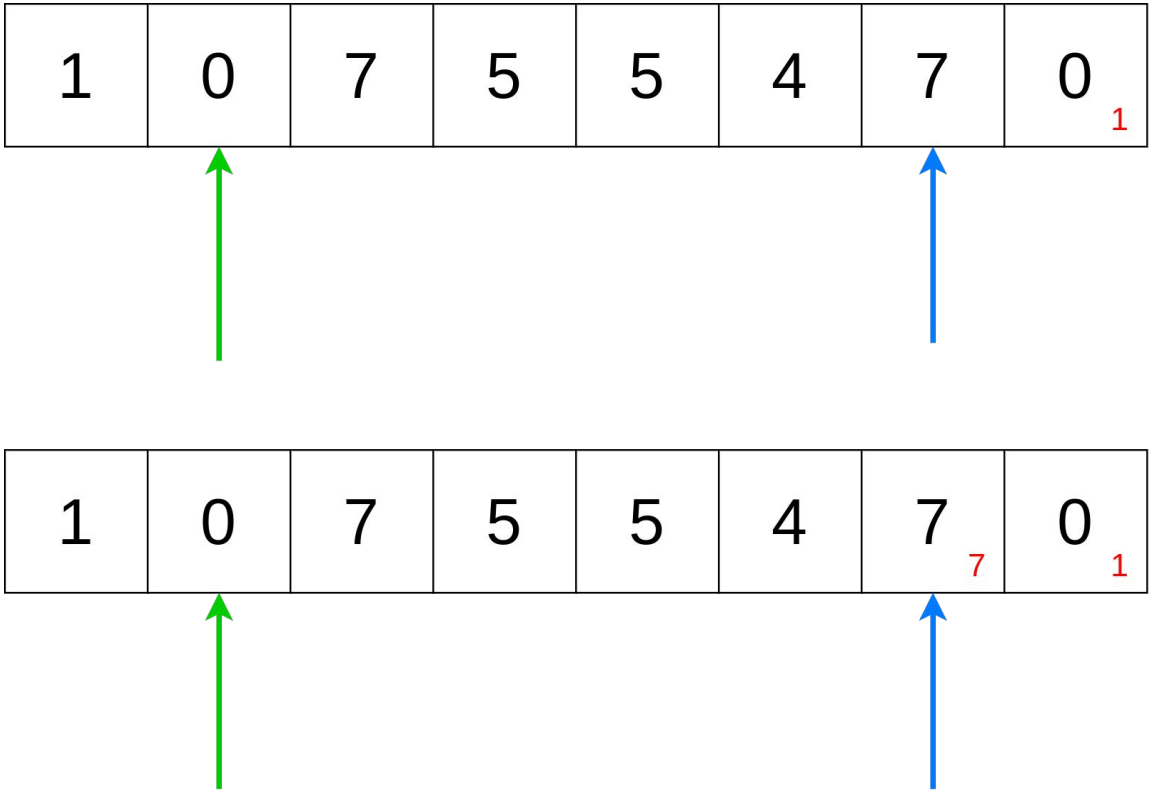
Sequential Algorithm

- In the following section, we will describe the implementation details of our sequential algorithm, which performs much better than the naive implementation.
- The main idea is very simple - we keep only one instance of the current number în memory.
- At each step, we go through the number starting from both ends and make additions of two “symmetrical” digits. We store the resulted digits in the same number.
- We call symmetrical digits those pairs of digits which are equally spaced from the beginning and end of the number.

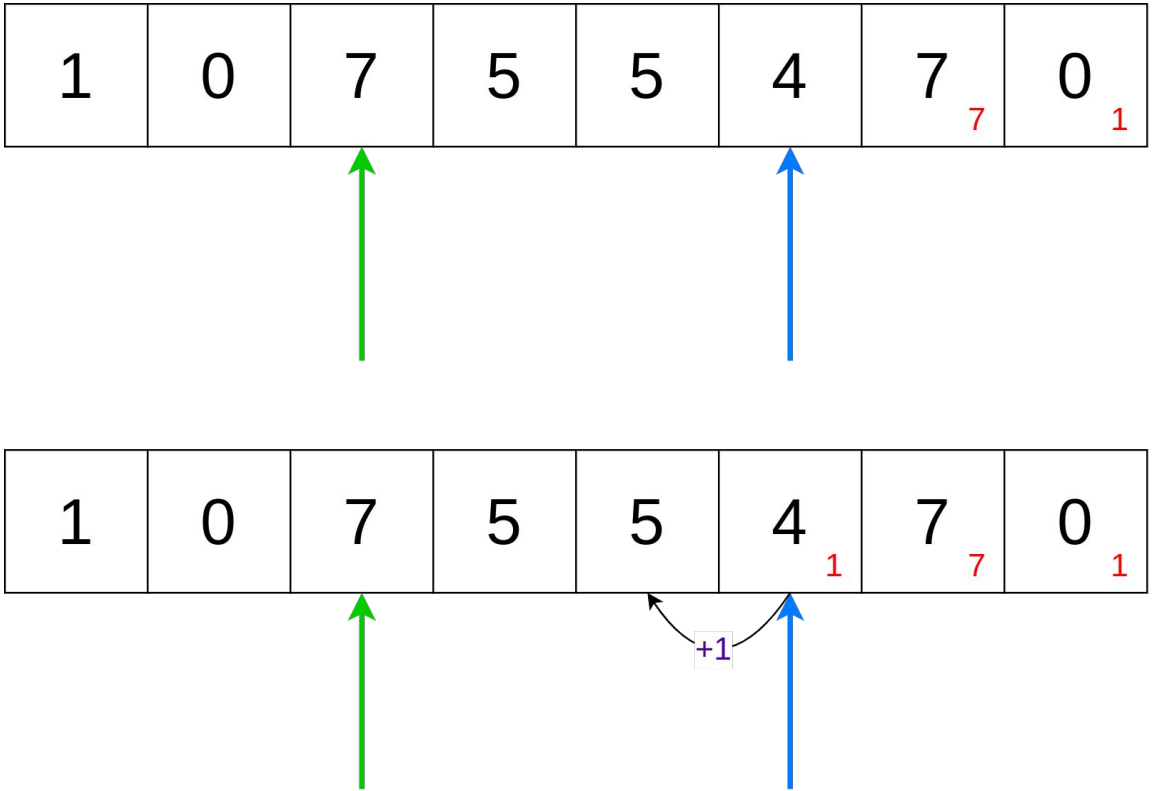
Visual Representation



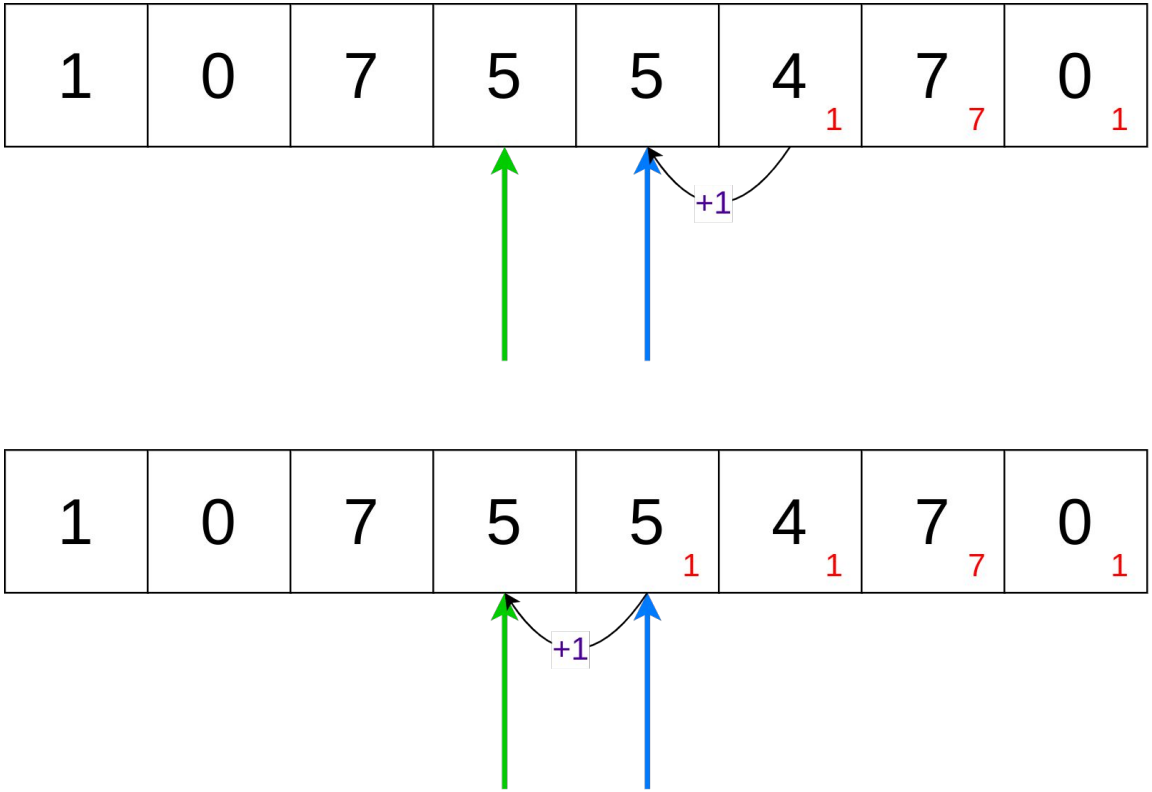
Visual Representation



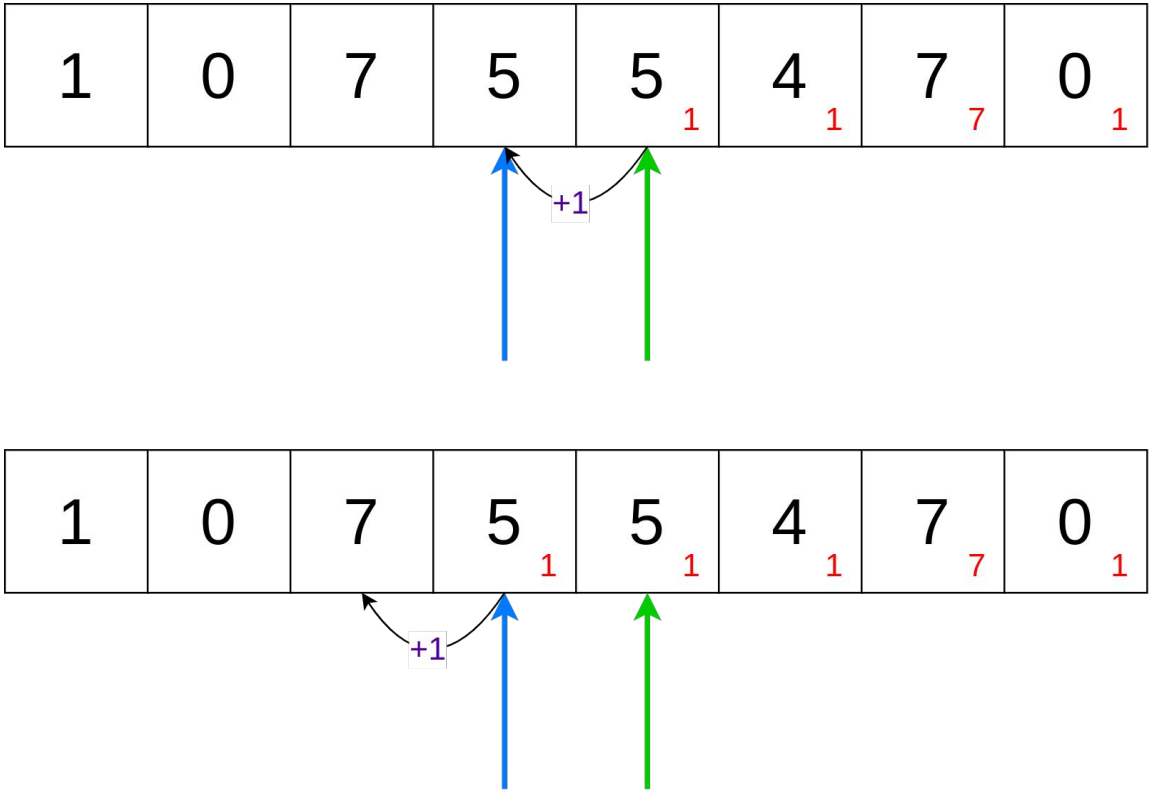
Visual Representation



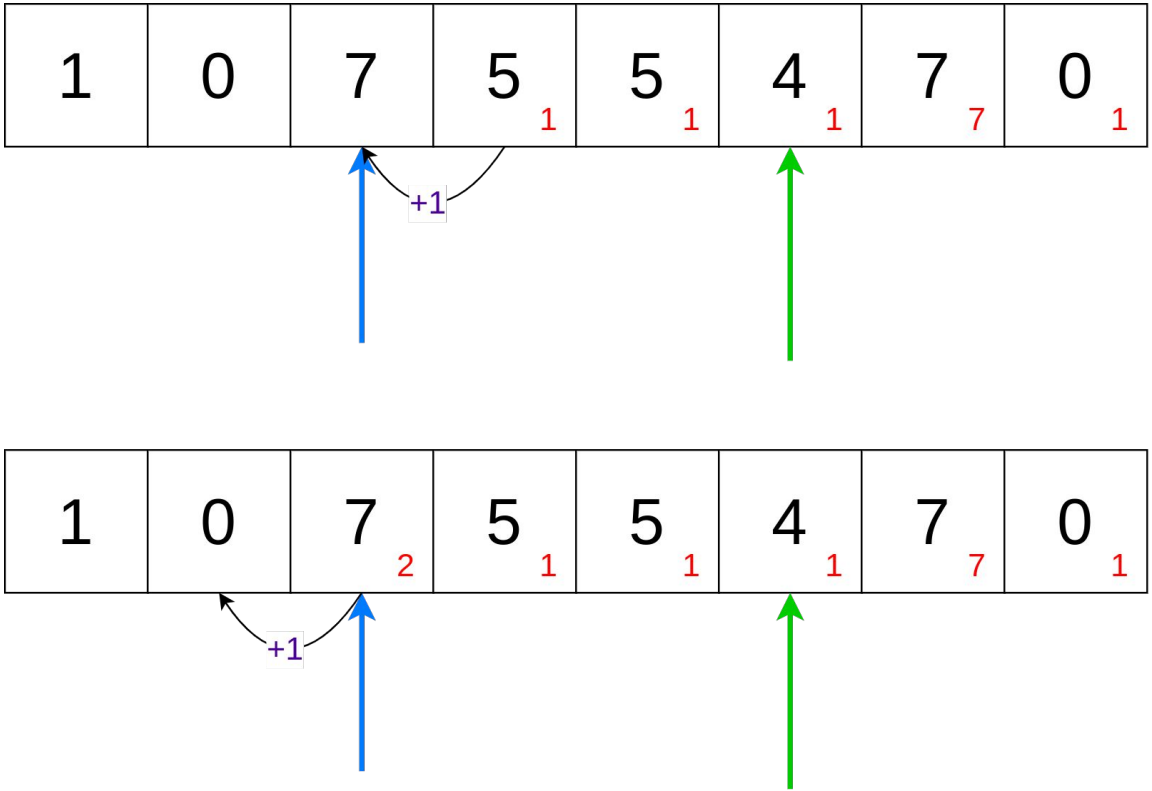
Visual Representation



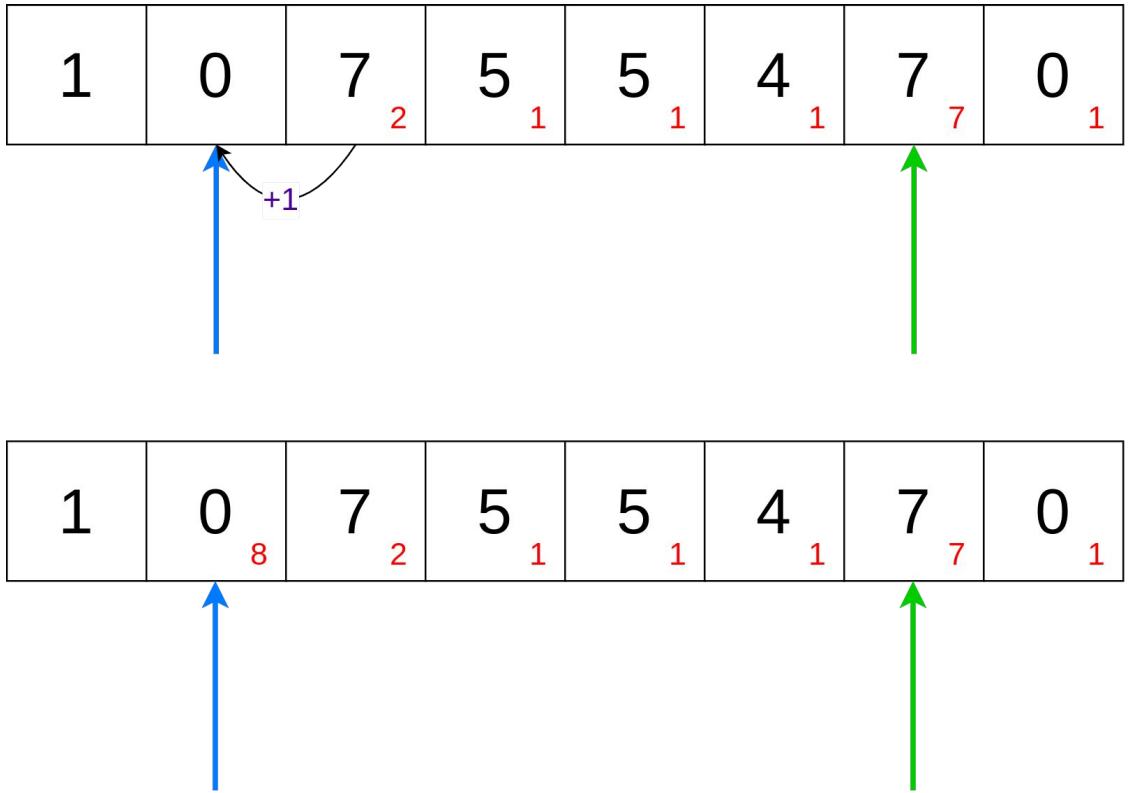
Visual Representation



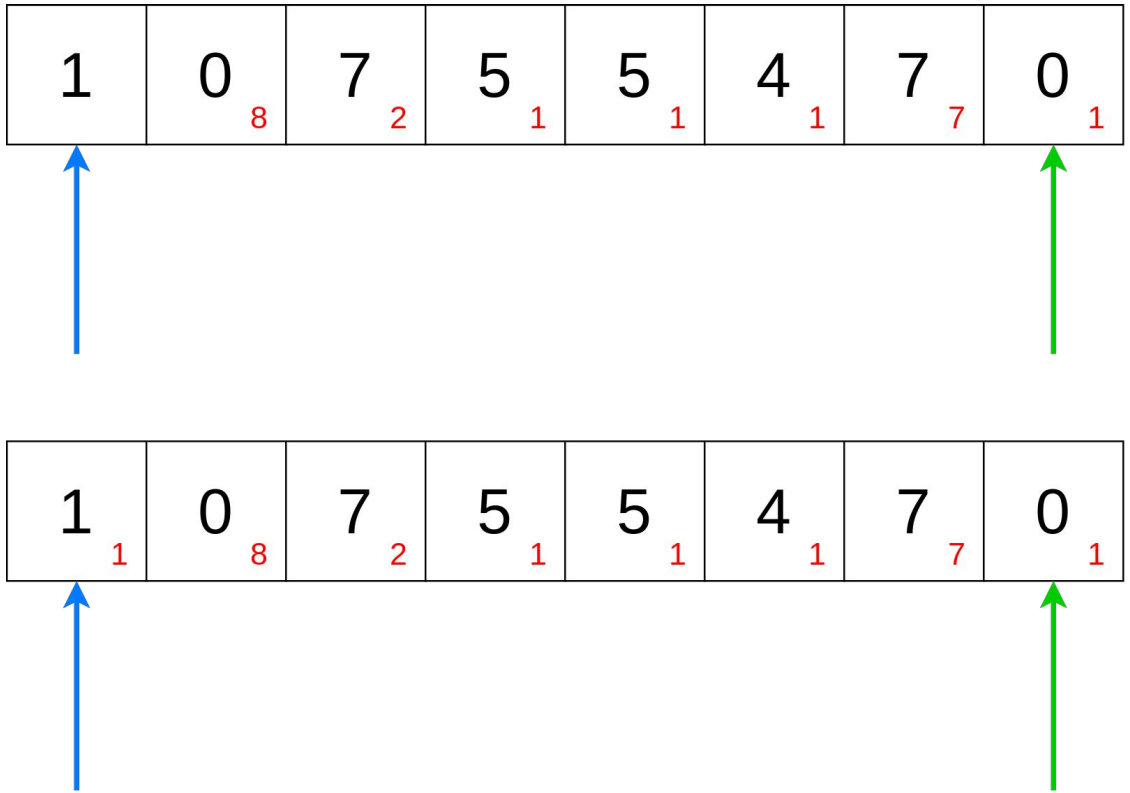
Visual Representation



Visual Representation

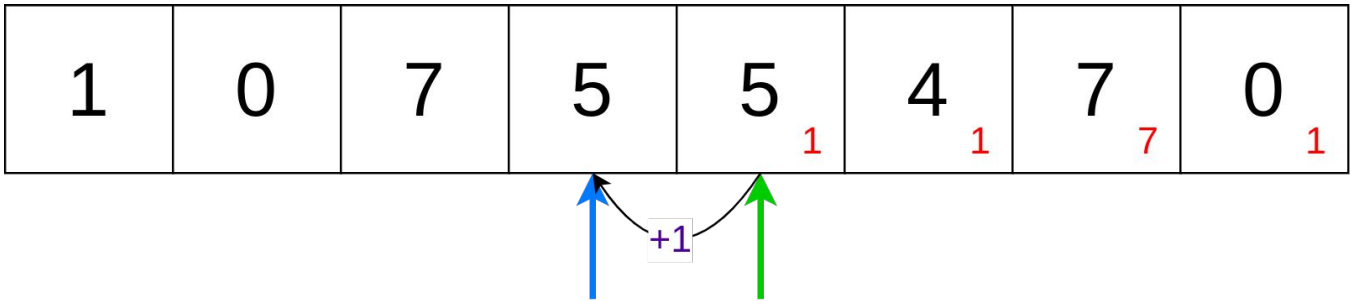


Visual Representation



Sequential Algorithm

- Why the new digits are represented with red and put near every corresponding digit in the same number?
- Can't we just overwrite the digits in the existing number?
- No, because as soon as we pass in the second half with the first iterator, the symmetric digit would already be overwritten by the new digit obtained at the current iteration.



Sequential Algorithm

- How do we store the new digits (represented with red) in the same number?
- The solution is based on a cool mathematical trick, which involves modular arithmetic.
- Suppose we have two natural numbers a and b.
- For every natural number $T \geq \max(a, b)$, a can be written as $(a + b*T) \% T$ and b can be written as $(a + b*T) / T$.
- In other words, we can “recover” both a and b from $T = a + b * T$. Number a is the remainder of the division and b is the quotient of the division.

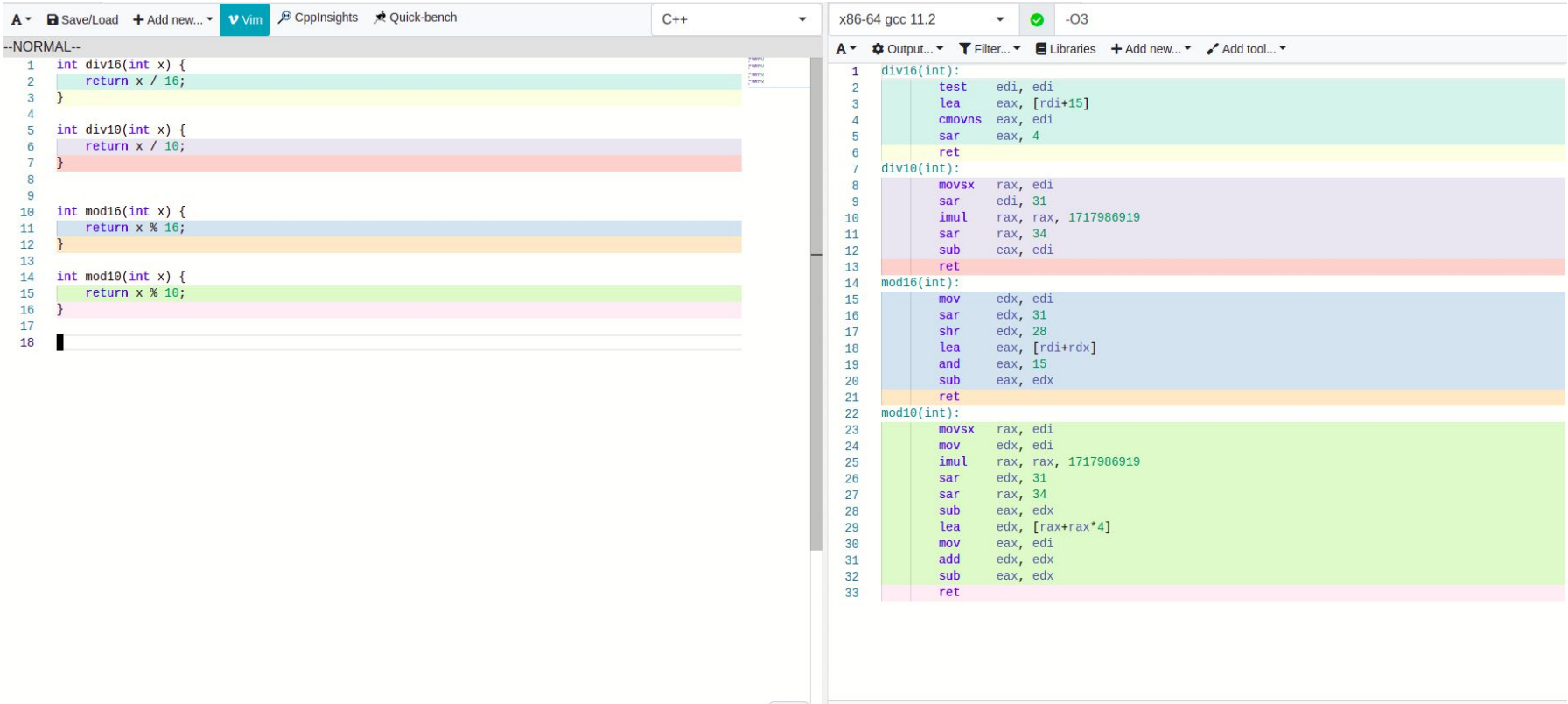
Sequential Algorithm

- Using this technique, we can store two digits in the same memory slot at the same time. Because we work with base-10 digits, we can choose any $T \geq 10$.
- Should we use 10 for our T value? Is there another value that would allow us to make calculations more efficiently?

Sequential Algorithm

- We know that the division instructions executed by the processor are extremely expensive (15-80 clock cycles per instruction depending on architecture). Modern compilers have optimization methods that translate divisions into smart multiplications (when the divisor is known at compile-time), but the multiplication also takes longer than the basic instructions.
- Can we choose a value for T in such a way that we “don’t need” any division / multiplication?
- Yes, we can choose a power of 2 as our value (for example 16), because we will no longer need divisions but shifts, which are much faster (1 single cycle per instruction).

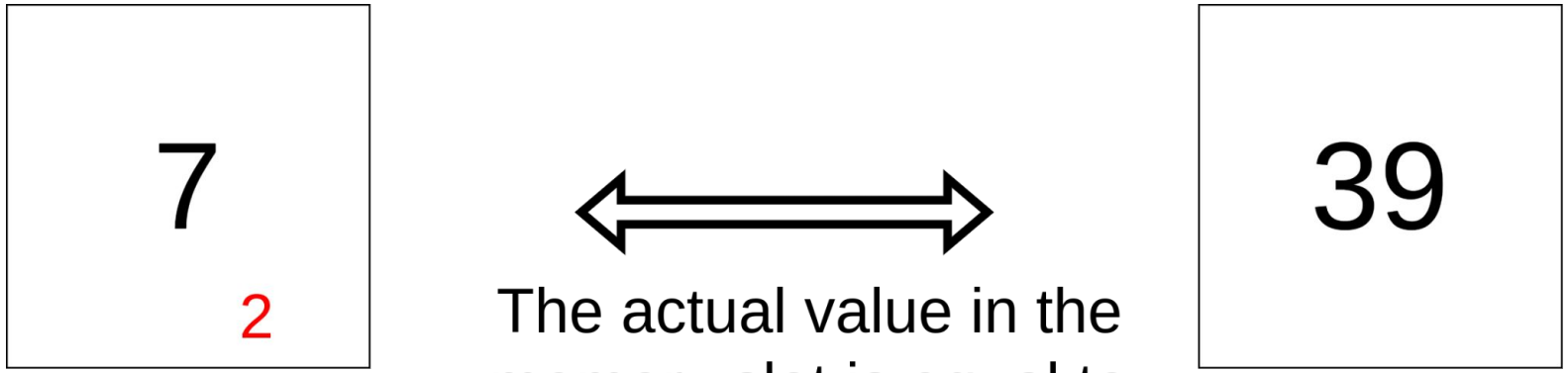
Sequential Algorithm



Sequential Algorithm

- Let’s talk about how each particular “digit” is stored in memory. Each “digit” is stored as a byte (8 bits). The one-byte memory slot can store 256 distinct values.
- As a consequence, that memory slot will be populated with the value that will hold both the value of the current digit and the value of the new digit at the same time.
- Because we chose T as 16, the maximum possible value fits into memory (the maximum value is 0x99 == 153)

Sequential Algorithm

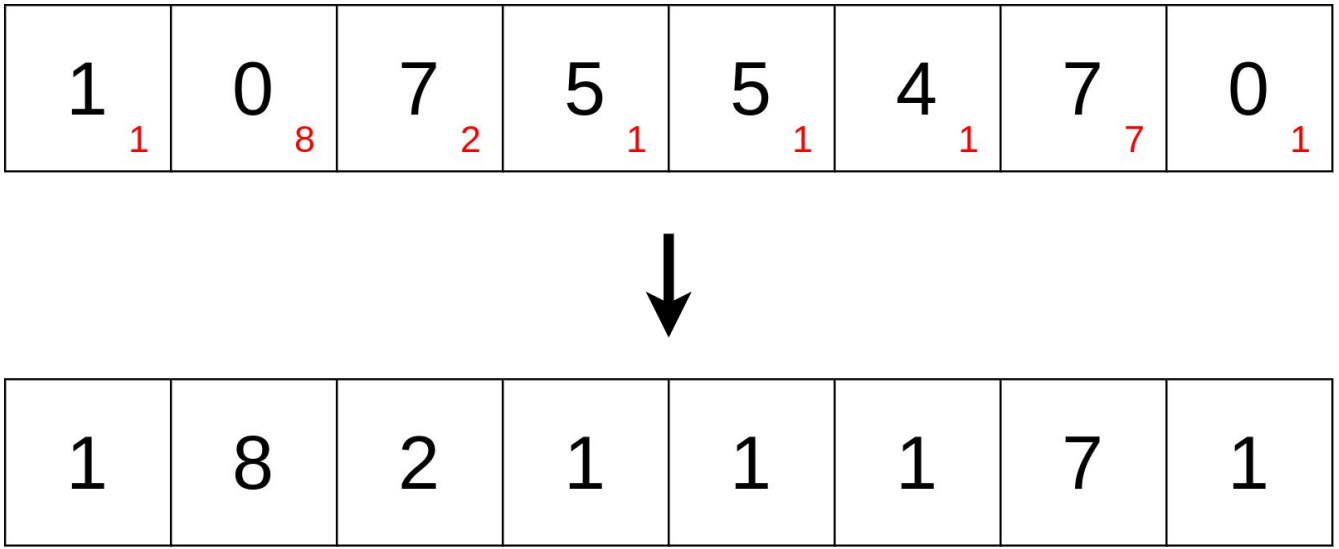


The actual value in the memory slot is equal to
a + b*T = 39

a = 7 (old digit)
b = 2 (new digit)
T = 16

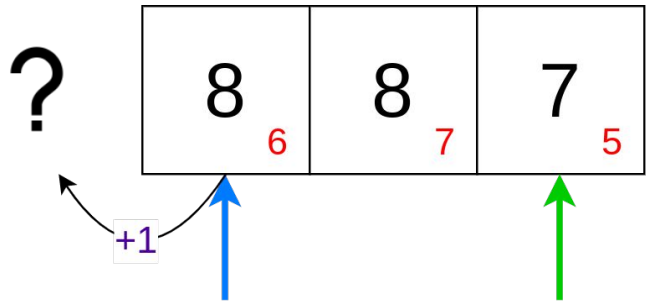
Sequential Algorithm

- After each iteration of the algorithm, we will go through each “digit” and update the value of the “old” digit. That’s because the new digit will become the old digit for the next iteration.



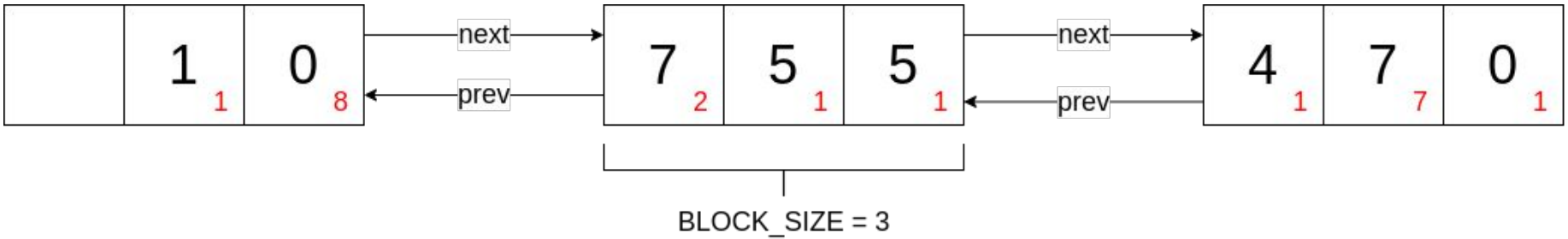
Sequential Algorithm

- So far, we haven't talked about how to store the list of digits in memory.
- A first idea would be to use a vector of bytes. But what size should we allocate for this vector?
- Also, we did not discuss the situation in which the addition would generate a number with more digits than the initial number (the last pair of digits yields a carry). Where should we put that particular digit?



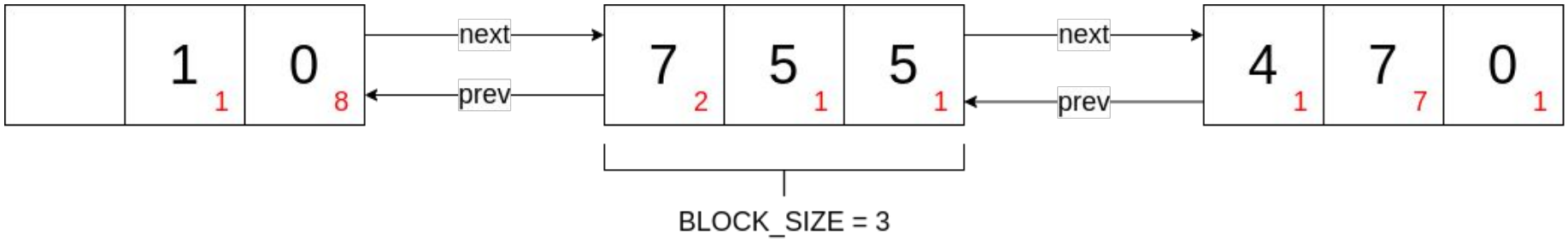
Sequential Algorithm

- Because of the issues that can occur using static memory allocation, we have built a solution that allows us to allocate dynamic memory efficiently.
- A doubly linked list of blocks - where each block is an array of fixed size holding digits.
- The first block will hold digits from the most significant one, while the last block will hold digits from the least significant one.



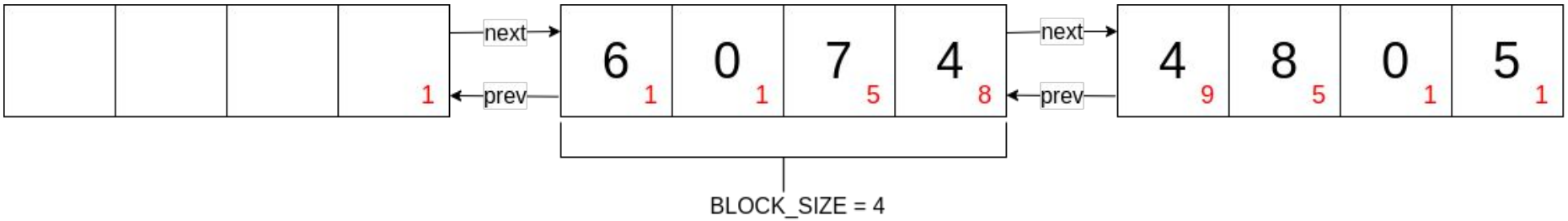
Sequential Algorithm

- When a new digit needs to be added to the number, the rightmost available slot of the first block will be used. If no slot is available, a new block will be created instead.



Pushing a new digit to an existing data block when free space is available

Sequential Algorithm

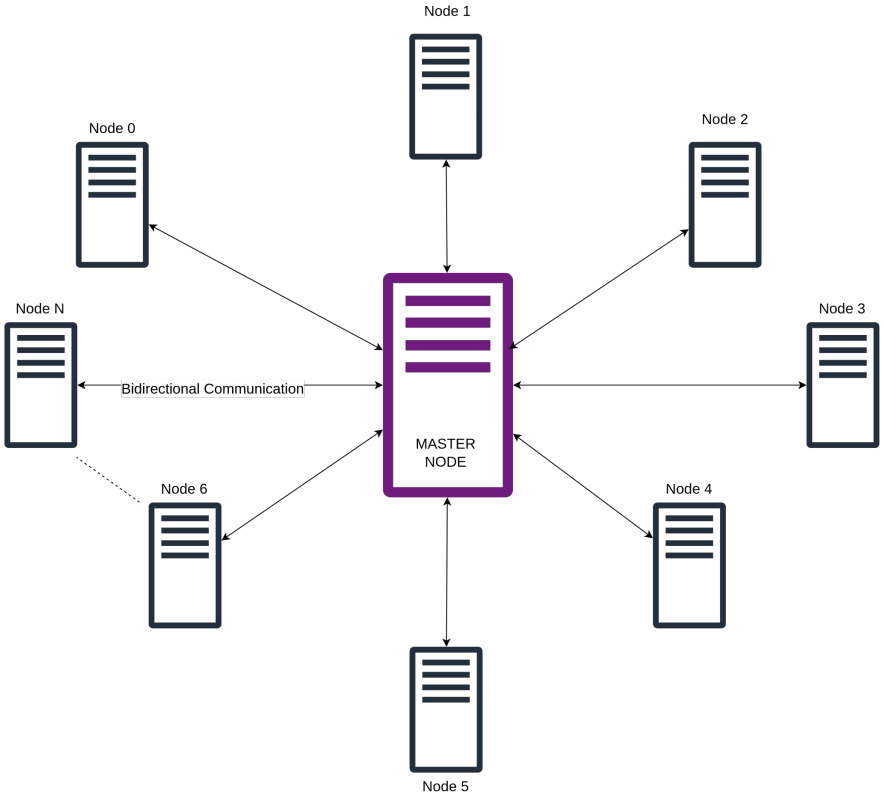


Pushing a new digit to a new data block

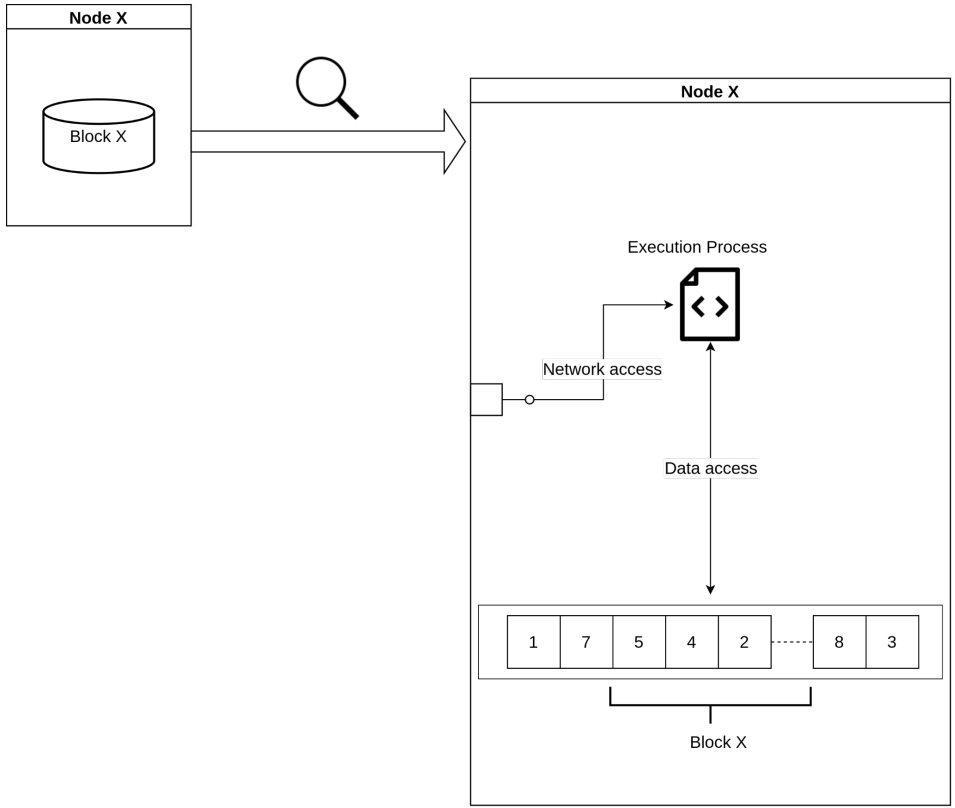
Case study for a distributed algorithm

- The above mentioned technique for storing the representation of a number is the first step ahead for designing a parallel / distributed algorithm.
- The data blocks may be distributed across a network of computers. Each node (computer) will have its own data block and make the required computation in parallel.
- However, communication between nodes is required. That’s because the digits from the last block must be added with the digits from the first block, and this blocks will be stored on different machines.

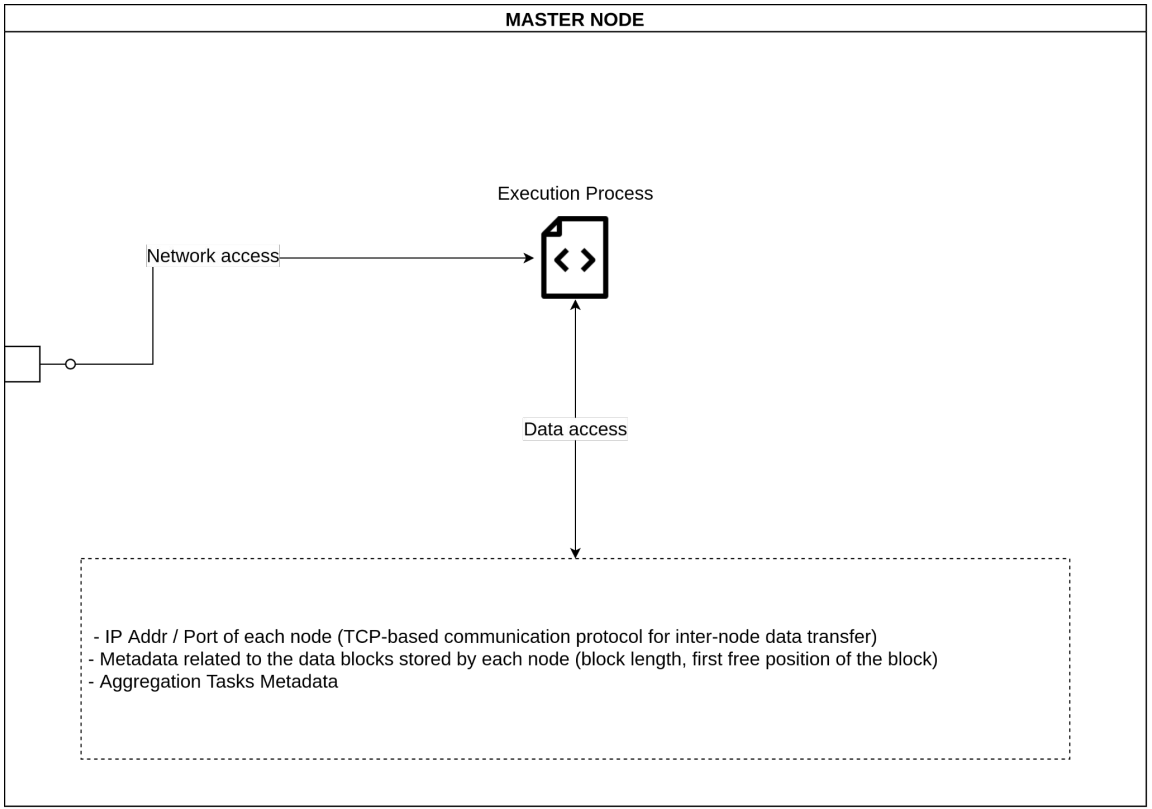
Case study for a distributed algorithm



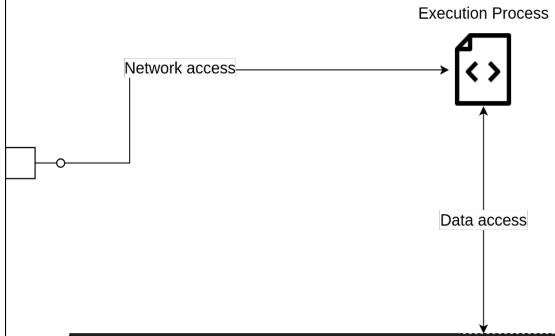
Case study for a distributed algorithm



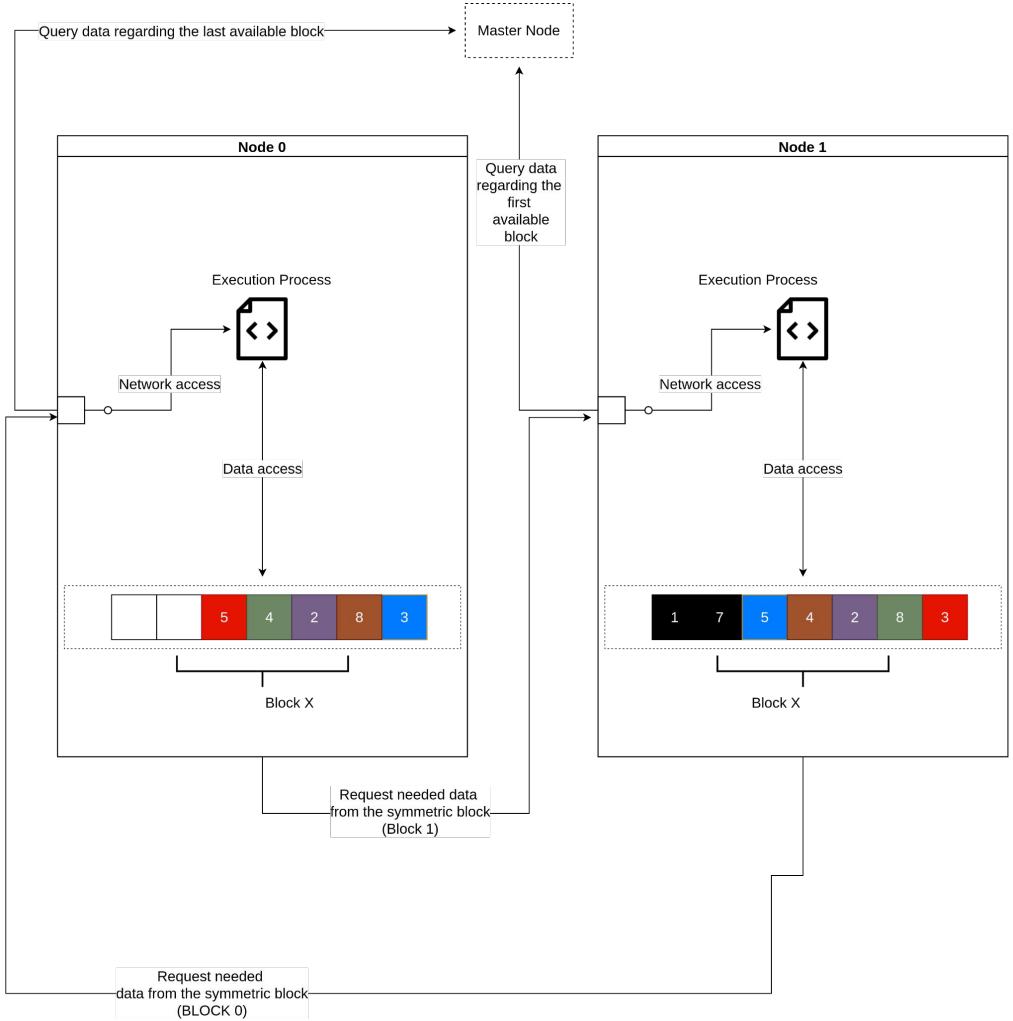
Case study for a distributed algorithm



MASTER NODE




```
{
    "nodes": [
        {
            "identifier": "NODE_0",
            "block_count": "1",
            "block_lenght": [10000],
            "block_first_free_position": [564],
            "public_ip": "53.62.14.223",
            "port": "80"
        },
        {
            "identifier": "NODE_1",
            "block_count": "1",
            "block_lenght": [10000],
            "block_first_free_position": [0],
            "public_ip": "83.44.32.115",
            "port": "4075"
        },
        {
            "identifier": "NODE_2",
            "block_count": "1",
            "block_lenght": [10000],
            "block_first_free_position": [0],
            "public_ip": "3.45.223.18",
            "port": "8081"
        }
    ]
}
```


Query data regarding the last available block
Master Node
Node 0
Node 1
Query data regarding the first available block
Execution Process
Execution Process
Network access
Network access
Data access
Data access
5 4 2 8 3
1 7 5 4 2 8 3
Block X
Block X
Request needed data from the symmetric block (Block 1)
Request needed data from the symmetric block (BLOCK 0)

ANY QUESTIONS?

Not done yet :). Approach a theoretical situation

- The solutions presented can be applied and written without any major flaws, but I'll invite you to answer a question: **Why is not that fast?**

Approach a theoretical situation

- The solutions presented can be applied and written without any major flaws, but I'll invite you to answer a question: **Why is not that fast?**
- **It's all because of RAM!**
- Now I'll invite you to think for a new solutions to the problem mentioned above.

Approach a theoretical situation

- What if instead of using 8 bits and memorizing each digit inside a char type, we will use an **uint64** type to memorize 8 “digits” inside a variabile of this type.

```
masks = np.array([
    0xFFFFFFF0, 0xFFFFFF0F, 0xFFFFF0FF, 0xFFFF0FFF, 0xFFF0FFFF, 0xFF0FFFFF, 0xF0FFFFFF, 0x0FFFFFFF,
    0xFFFFFFF1, 0xFFFFFF1F, 0xFFFFF1FF, 0xFFFF1FFF, 0xFFF1FFFF, 0xFF1FFFFF, 0xF1FFFFFF, 0x1FFFFFFF,
    0xFFFFFFF2, 0xFFFFFF2F, 0xFFFFF2FF, 0xFFFF2FFF, 0xFFF2FFFF, 0xFF2FFFFF, 0xF2FFFFFF, 0x2FFFFFFF,
    0xFFFFFFF3, 0xFFFFFF3F, 0xFFFFF3FF, 0xFFFF3FFF, 0xFFF3FFFF, 0xFF3FFFFF, 0xF3FFFFFF, 0x3FFFFFFF,
    0xFFFFFFF4, 0xFFFFFF4F, 0xFFFFF4FF, 0xFFFF4FFF, 0xFFF4FFFF, 0xFF4FFFFF, 0xF4FFFFFF, 0x4FFFFFFF,
    0xFFFFFFF5, 0xFFFFFF5F, 0xFFFFF5FF, 0xFFFF5FFF, 0xFFF5FFFF, 0xFF5FFFFF, 0xF5FFFFFF, 0x5FFFFFFF,
    0xFFFFFFF6, 0xFFFFFF6F, 0xFFFFF6FF, 0xFFFF6FFF, 0xFFF6FFFF, 0xFF6FFFFF, 0xF6FFFFFF, 0x6FFFFFFF,
    0xFFFFFFF7, 0xFFFFFF7F, 0xFFFFF7FF, 0xFFFF7FFF, 0xFFF7FFFF, 0xFF7FFFFF, 0xF7FFFFFF, 0x7FFFFFFF,
    0xFFFFFFF8, 0xFFFFFF8F, 0xFFFFF8FF, 0xFFFF8FFF, 0xFFF8FFFF, 0xFF8FFFFF, 0xF8FFFFFF, 0x8FFFFFFF,
    0xFFFFFFF9, 0xFFFFFF9F, 0xFFFFF9FF, 0xFFFF9FFF, 0xFFF9FFFF, 0xFF9FFFFF, 0xF9FFFFFF, 0x9FFFFFFF,
  ])
```

Approach a theoretical situation

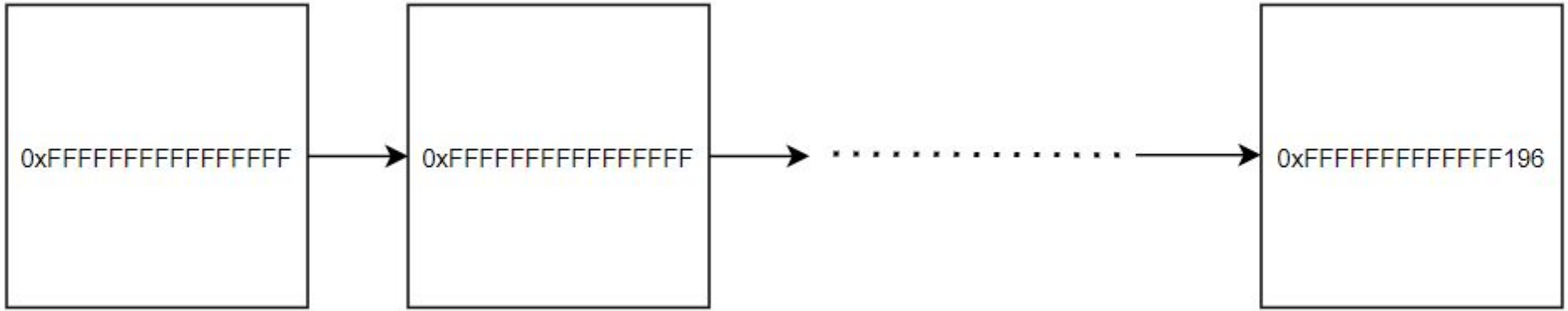
- Here is a small demo of how we can convert some of the decimal digits using some **masks**.
- **But can you see a breach in this situation?**
- A small hint we'll be related to the number of digits increasing through each iteration.

```
insertable_number = np.uint64(0xFFFFFFFF);
current_position = 0
while (number): #let's say the number=196 or other big number generated
    insertable_number &= np.uint64(masks[(number % 10) * 8 + current_position])

    current_position += 1
    number //= 10
```

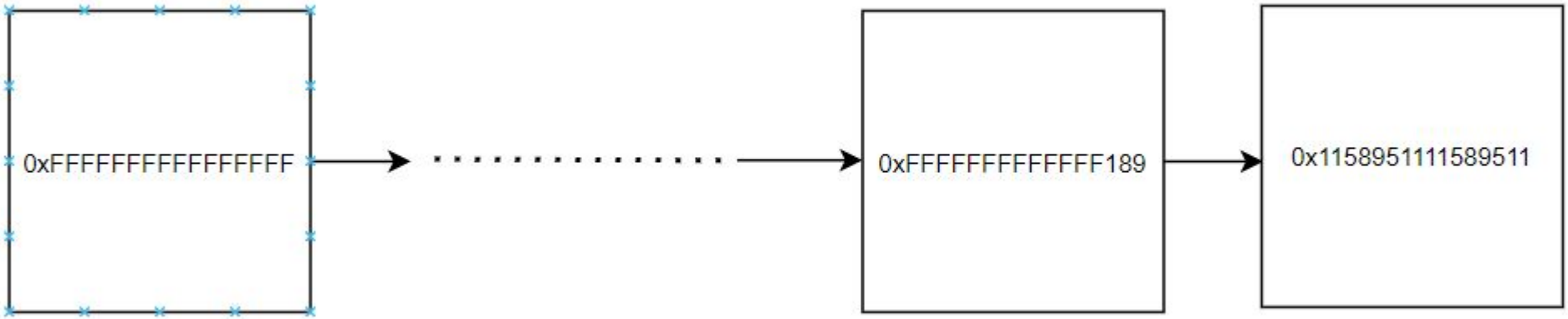
Approach a theoretical situation

- The number **196** represented in memory with this solution.



Approach a theoretical situation

- Well the whole story is about how we can get a clear view on how to manage the inverse of the number when is not represented in a single node of the list.
- Is not impossible, but is hard to manage.



Approach a theoretical situation

- Let’s say that we managed to get the inverse of the number now, what can we do with the addition algorithm?

Approach a theoretical situation

- Let’s say that we managed to get the inverse of the number now, what can we do with the addition algorithm?
- But what if we parallelize the addition algorithm, and do each summation of two digits on a separate thread.
- **What problems can appear using this method?**

Approach a theoretical situation

- Well a thread is like a lambda, which means it doesn't take care of other dependencies in this case we can have a digit which is computed in the left-most part faster than the one in the right-most part.
- **The problem that we'll have is with the carry digit of each individual summation.**
- But even for this we have a solution. How about changing the base yet again in order to represent this digits. **But what base?**

Approach a theoretical situation

- Well by intuition we can take the greatest digits in the base 10 and thus by adding them together we obtain **18** so we can conclude that the base in which we can represent the digits is **19**? **What do you think?**

Approach a theoretical situation

- Well by intuition we can take the greatest digits in the base 10 and thus by adding them together we obtain **18** so thus we can conclude that the base in which we can represent the digits is **19**?
- **WRONG! Don’t forget the carry digit, so in total we will have a maximum value of 19, the base will be 20**.
- We prove by induction that the maximum carry is 1 in any base *b*. This is clearly true when carrying over units to the b^1 place, because the maximum is $(b - 1) + (b - 1) = 1(b - 2)_b$. Suppose by induction that we are adding the digits in the b^k place, let’s say *x* and *y* with $x, y < b$, and let *c* be the carry from b^{k-1}, which is at most 1.

Approach a theoretical situation

- Then the result $x + y + c \leq (b - 1) + (b - 1) + 1 = 1(b - 1)_b$, so the maximum carry to the b^{k+1} place is a 1, and the result follows by induction.
- Looking at the base **10** case, that means the largest “digit” sum you can get is **19**, which is what happens when you carry over a **1** and have a **9** for each digit.
- The maximum in general is ***2b - 1***, which has a 1 in the b^1 place.
- Coming back to the problem, after we do the addition we will need to solve some “conflicts”, and this concludes an iteration.

THE END