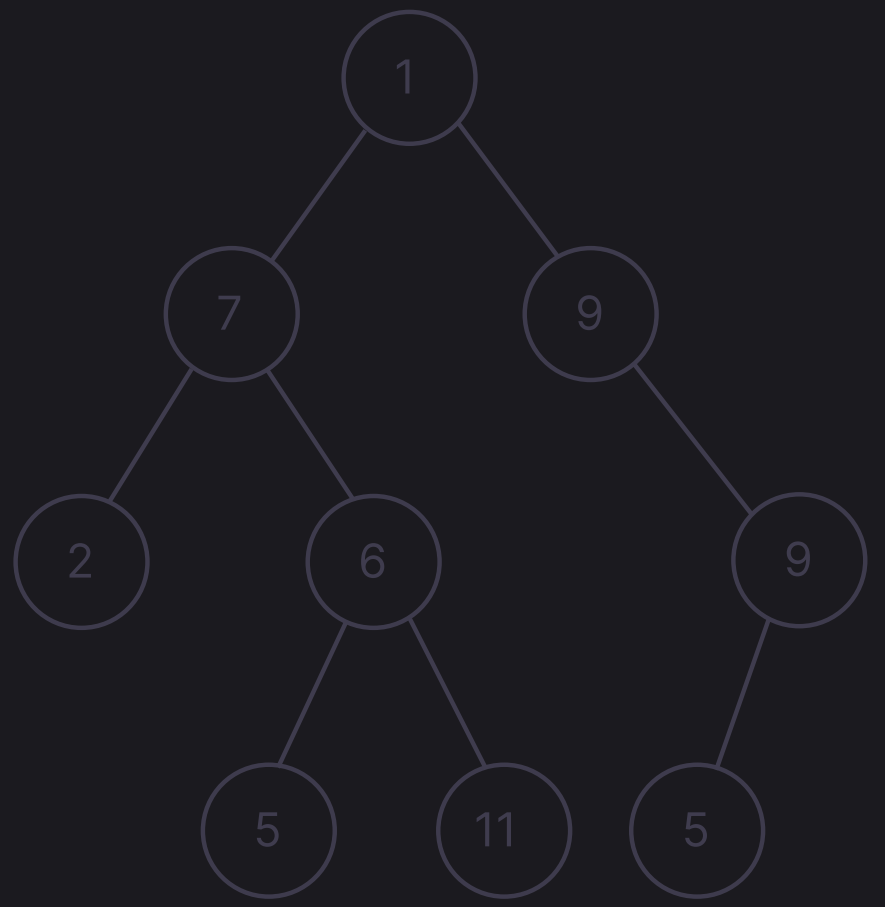
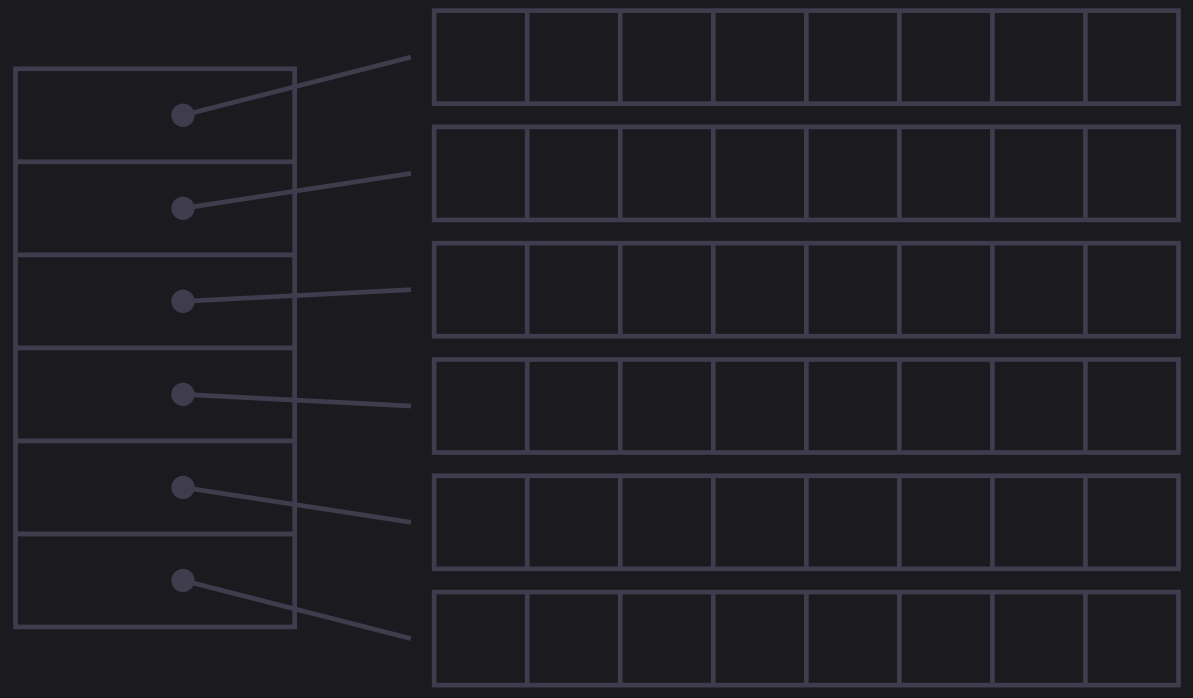
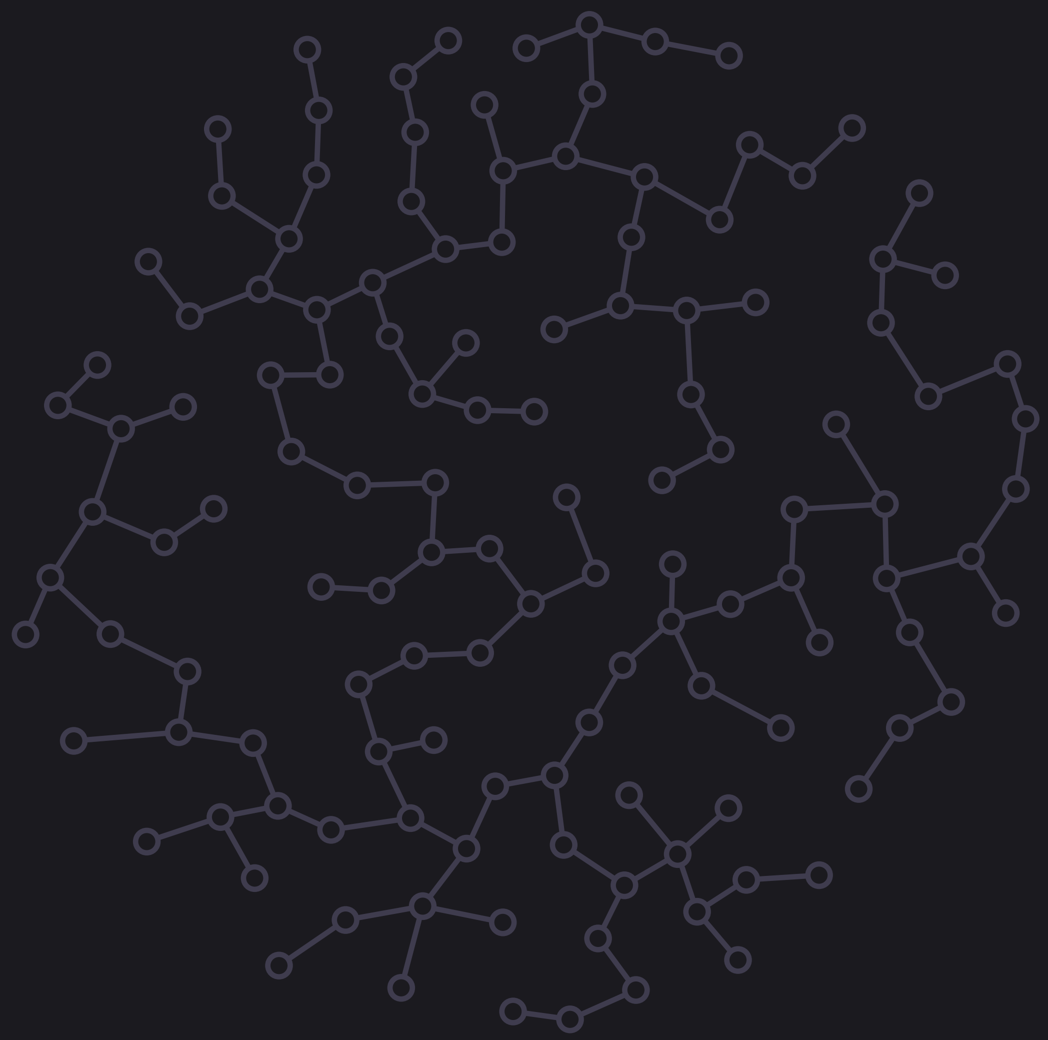
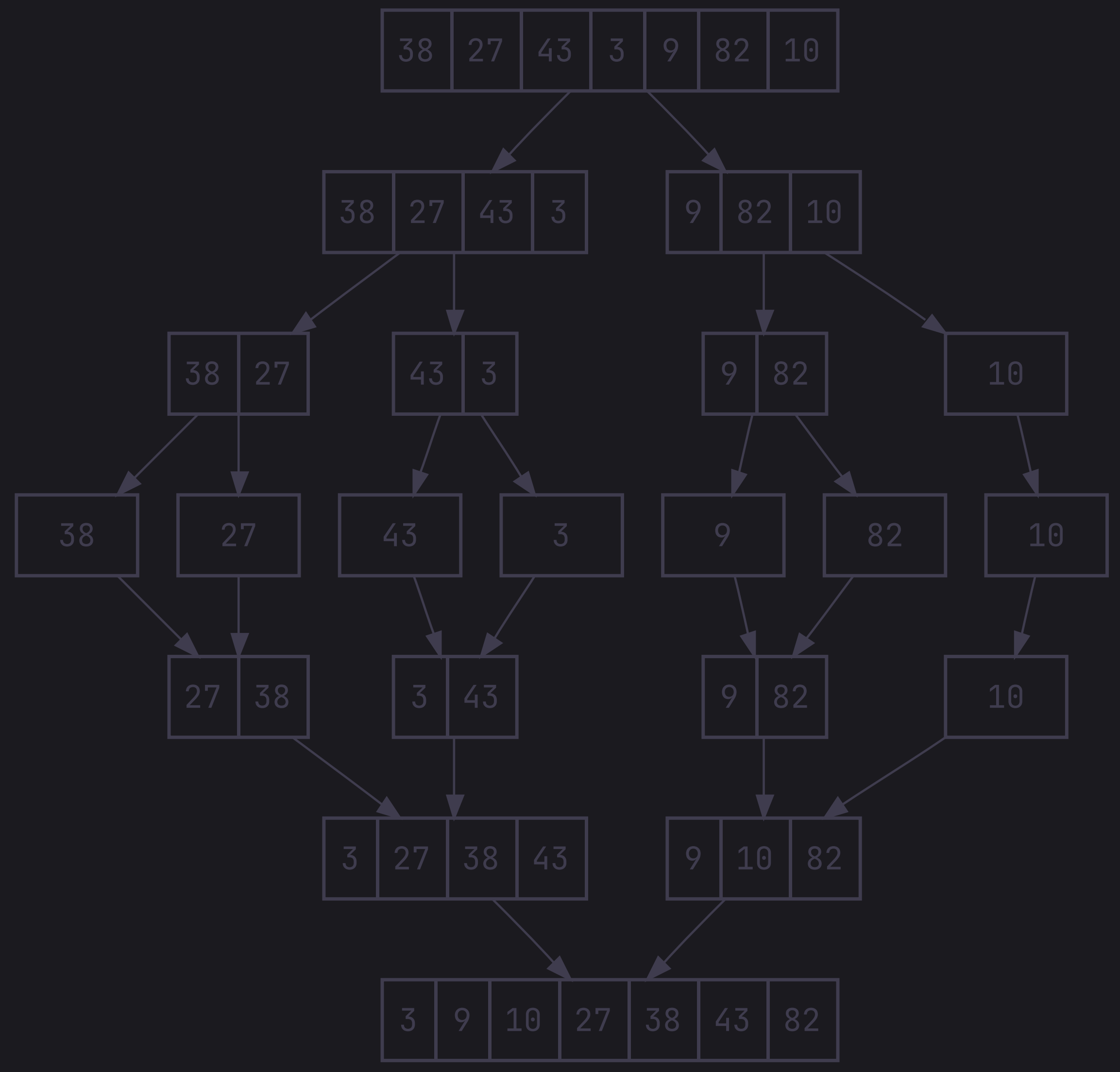




data structures & algorithms

Tutorial 2





🔥🙋‍♀️

Burning questions from last week?

This week's lab

it's an array of tennis balls

This week we are learning all about **vector** (an array like data structure)

- Playing with std::vector
- Our first leetcode problem
- Building our own toy version of vector

std::vector

```cpp
std::vector<int> nums { 3, 1, 4, 1, 5, 9 };
```

nums	3	1	4	5	1	9
	0	1	2	3	4	5

- A vector is like a raw array
- It can store any data you like
- but it also **resizes automatically**

Vector types

```cpp
std::vector<char> letters { "T", "A", "C", "O" };
```

letters	"T"	"A"	"C"	"O"
	0	1	2	3

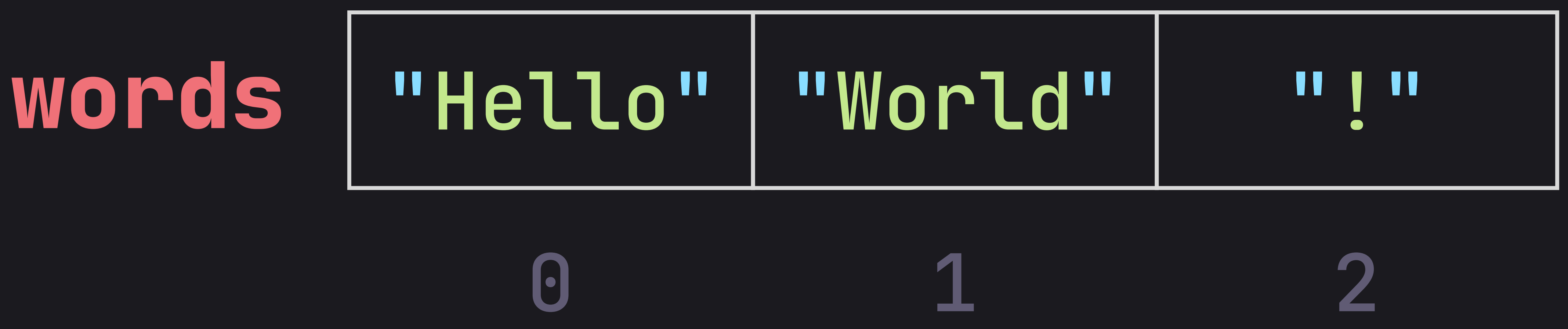
Vector types

```cpp
std::vector<bool> booleans { true, false, false, true };
```

booleans	true	false	false	true
	0	1	2	3

Vector types

```
std::vector<std::string> words { "Hello", "World", "!"  };
```



Size of a vector

```cpp
std::vector<int> nums { 3, 1, 4, 1, 5, 9 };
nums.size(); // returns: 6
```

size tells us how many elements there are in the vector

Accessing elements of a vector

```cpp
std::vector<int> nums { 3, 1, 4, 1, 5, 9 };
nums[2]; // returns: 4
```

We can access elements using the indexing operator []

Accessing elements of a vector

```cpp
std::vector<int> nums { 3, 1, 4, 1, 5, 9 };
nums.at(2); // returns: 4
```

nums	3	1	4	5	1	9
	0	1	2	3	4	5

We can also use the at method
to access elements

Looping over a vector

```cpp
std::vector<int> nums { 3, 1, 4, 1, 5, 9 };

for (int i = 0; i < nums.size(); i++) {
    std::cout << nums[i] << ", ";
}

// This prints out 3, 1, 4, 1, 5, 9,
```

Looping over a vector

```
std::vector<int> nums { 3, 1, 4, 1, 5, 9 };

for (std::size_t i = 0; i < nums.size(); i++) {
    std::cout << nums[i] << ", ";
}

// This prints out 3, 1, 4, 1, 5, 9,
```

`std::size_t` is always ≥ 0 (never negative)

`std::size_t` also allows for larger numbers than `int`

Range based for loop

```cpp
std::vector<int> nums { 3, 1, 4, 1, 5, 9 };

for (int number: nums) {
    std::cout << number << ", ";
}

// This prints out 3, 1, 4, 1, 5, 9,
```

Range based for loop

```cpp
std::vector<int> nums { 3, 1, 4, 1, 5, 9 };

for (int whatever: nums) {
    std::cout << whatever << ", ";
}


// This prints out 3, 1, 4, 1, 5, 9,
```

Adding elements to a vector

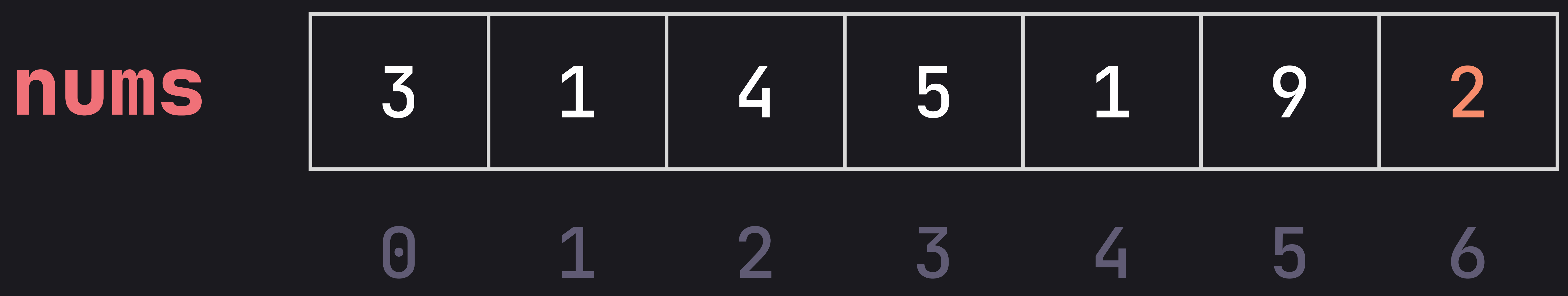
```cpp
std::vector<int> nums { 3, 1, 4, 1, 5, 9 };
```

nums	3	1	4	5	1	9
	0	1	2	3	4	5

We can “push” elements to the back/end of the vector using push_back

Adding elements to a vector

```
std::vector<int> nums { 3, 1, 4, 1, 5, 9 };
nums.push_back(2);
```



We can “push” elements to the back/end of the vector using push_back

Adding elements to a vector

```
std::vector<int> nums { 3, 1, 4, 1, 5, 9 };
nums.push_back(2);
nums.push_back(6);
```

nums	3	1	4	5	1	9	2	6
	0	1	2	3	4	5	6	7

We can “push” elements to the back/end of the vector using push_back

How a vector works

```
std::vector<int> nums { 3, 1, 4, 1, 5, 9 };
nums.push_back(2);
nums.push_back(6);
```

Adding elements to a raw array is expensive because it has a fixed size and so we need to **copy all the elements** to a larger array

Vector solves this by **reducing the number of times** it needs to increase the size of the array

How a vector works

```cpp
std::vector<int> nums { 3, 1, 4, 1, 5, 9 };
nums.size();      // returns 6
nums.capacity(); // returns 6
```

The size is how many elements are in the vector

The capacity is how much spare room there is in the vector.
(How many elements it can contain before we need to resize)

How a vector works

```cpp
std::vector<int> nums { 3, 1, 4, 1, 5, 9 };
→ nums.push_back(2);
nums.push_back(6);
```

nums

3	1	4	5	1	9

🛑 There is no more room in the vector!

i.e. size == capacity

How a vector works

```
std::vector<int> nums { 3, 1, 4, 1, 5, 9 };
→ nums.push_back(2);
nums.push_back(6);
```

So we need to allocate more memory and copy the data over

Notice we double the capacity of the vector

How a vector works

```
   std::vector<int> nums { 3, 1, 4, 1, 5, 9 };
⟶ nums.push_back(2);
   nums.push_back(6);
```

nums	3	1	4	5	1	9	2	?	?	?	?	?
	0	1	2	3	4	5	6	7	8	9	10	11

Now that we have space, we can add the 2

How a vector works

```
std::vector<int> nums { 3, 1, 4, 1, 5, 9 };
nums.push_back(2);
nums.push_back(6);
```

How a vector works

```
    std::vector<int> nums { 3, 1, 4, 1, 5, 9 };
    nums.push_back(2);
→   nums.push_back(6);
```

✓ There is more room in the vector!

i.e. size < capacity

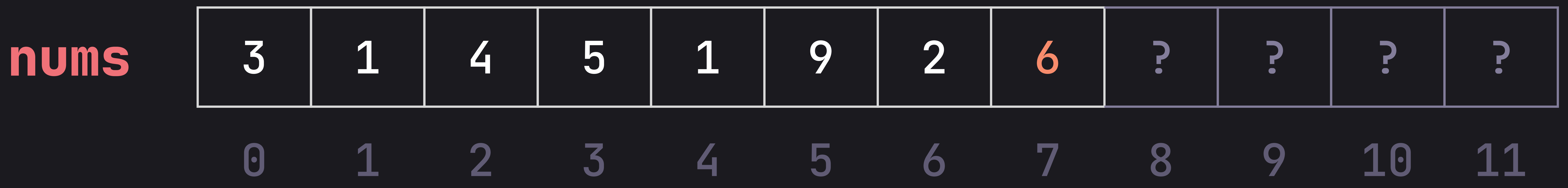
nums

3	1	4	5	1	9	2	6	?	?	?	?
0	1	2	3	4	5	6	7	8	9	10	11

So we can insert the 6 without needing to resize the vector

Capacity ≠ Size

```
nums.size();     // returns: 8
nums.capacity(); // returns: 12
```



Give the second activity a go!

we’ll come back to the first activity in 5 minutes

Our first leet code problem

Product of Array Except Self

Given an input **std::vector<int> nums**, the goal is to return a **std::vector<int> result** of the **same size** as **nums** such that **result[i]** is the product of all the integers in **nums** except for the **i**th one.

Product of Array Except Self

nums	3	1	4	5	1	9
	0	1	2	3	4	5

result	180	540	135	108	540	60
	0	1	2	3	4	5

Goal: return a vector of the same size as nums such that result[i] is the product of all the integers in nums except for the ith one.

Product of Array Except Self

nums

3	1	4	5	1	9
0	1	2	3	4	5

size = 6

result

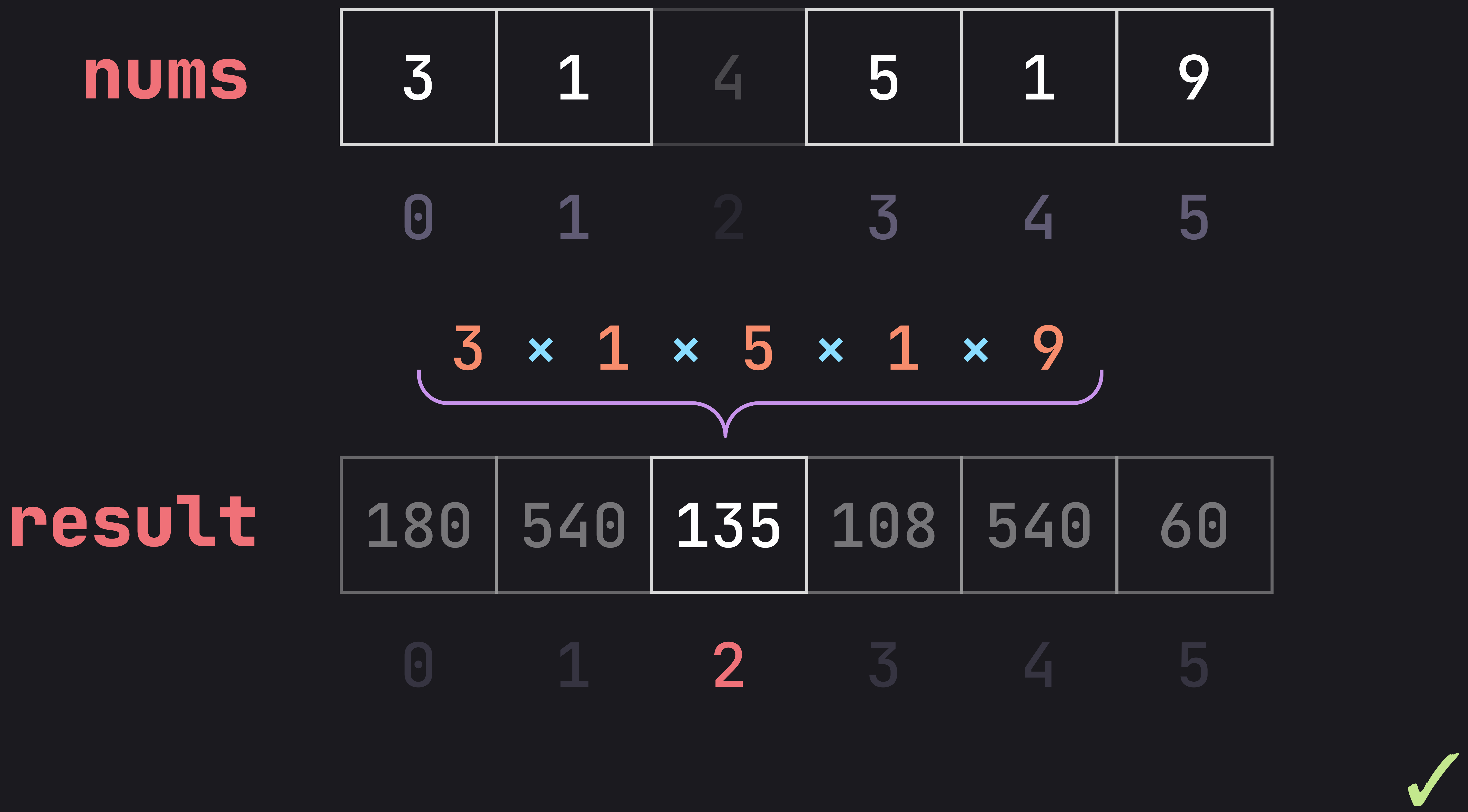
180	540	135	108	540	60
0	1	2	3	4	5

size = 6

✓

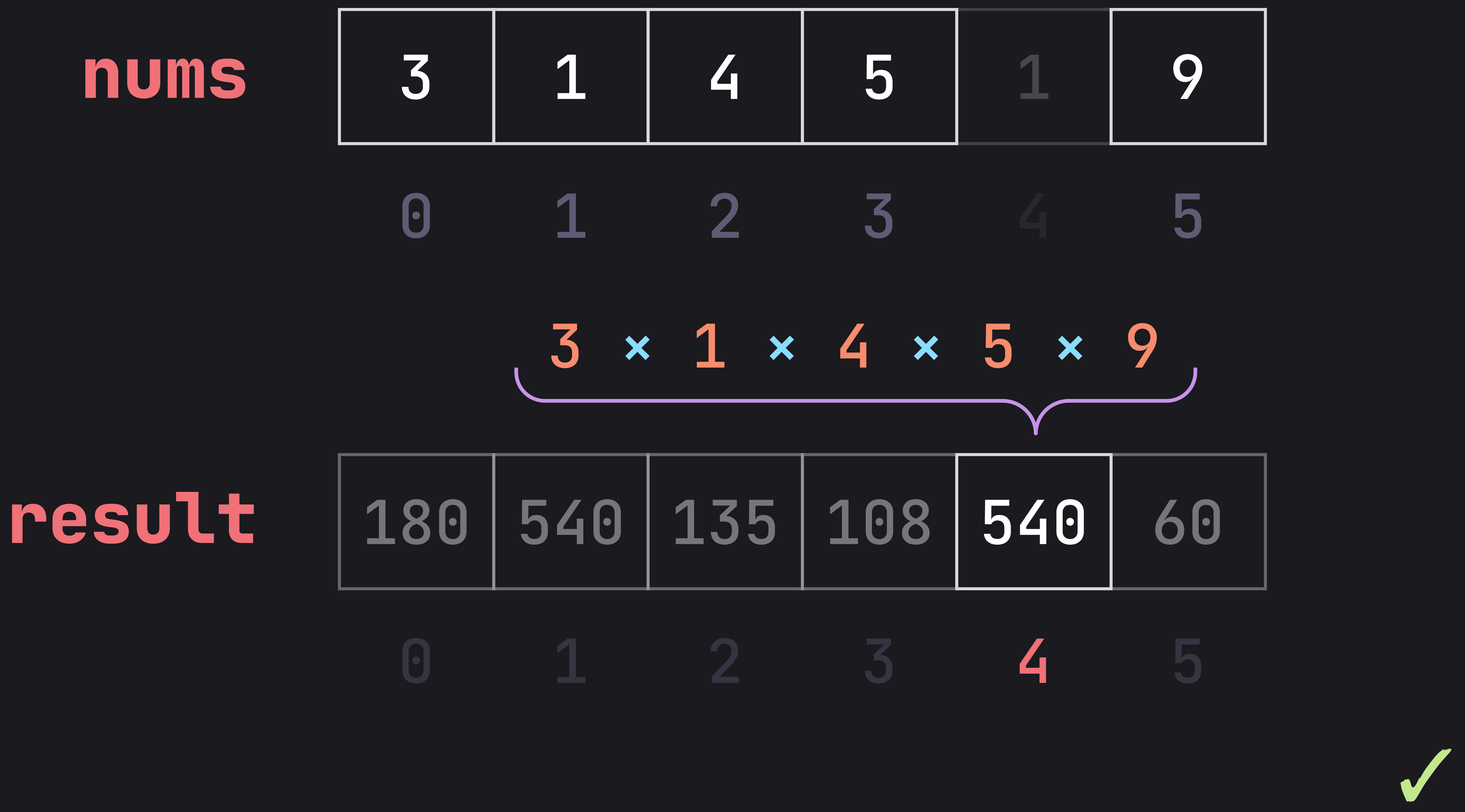
Goal: **return a vector of the same size** as nums such that result[i] is the product of all the integers in nums except for the ith one.

Product of Array Except Self



Goal: return a vector of the same size as nums such that **result[i] is the product of all the integers in nums except for the ith one**.

Product of Array Except Self



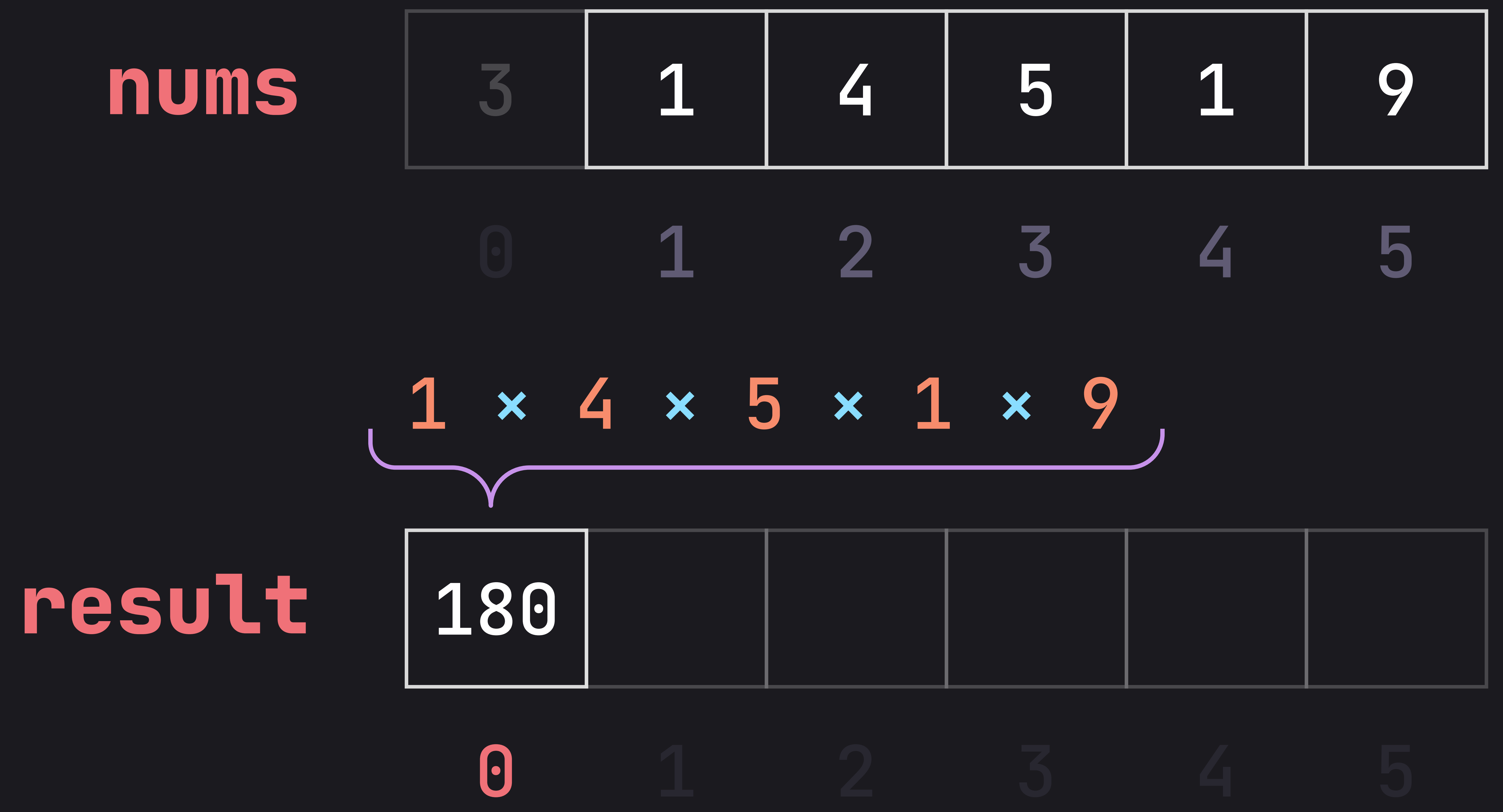
Goal: return a vector of the same size as nums such that **result[i] is the product of all the integers in nums except for the ith one**.

Any suggestions for a **brute force** algorithm? 🤔

A brute force algorithm is always a good *first step* when solving an algorithms problem

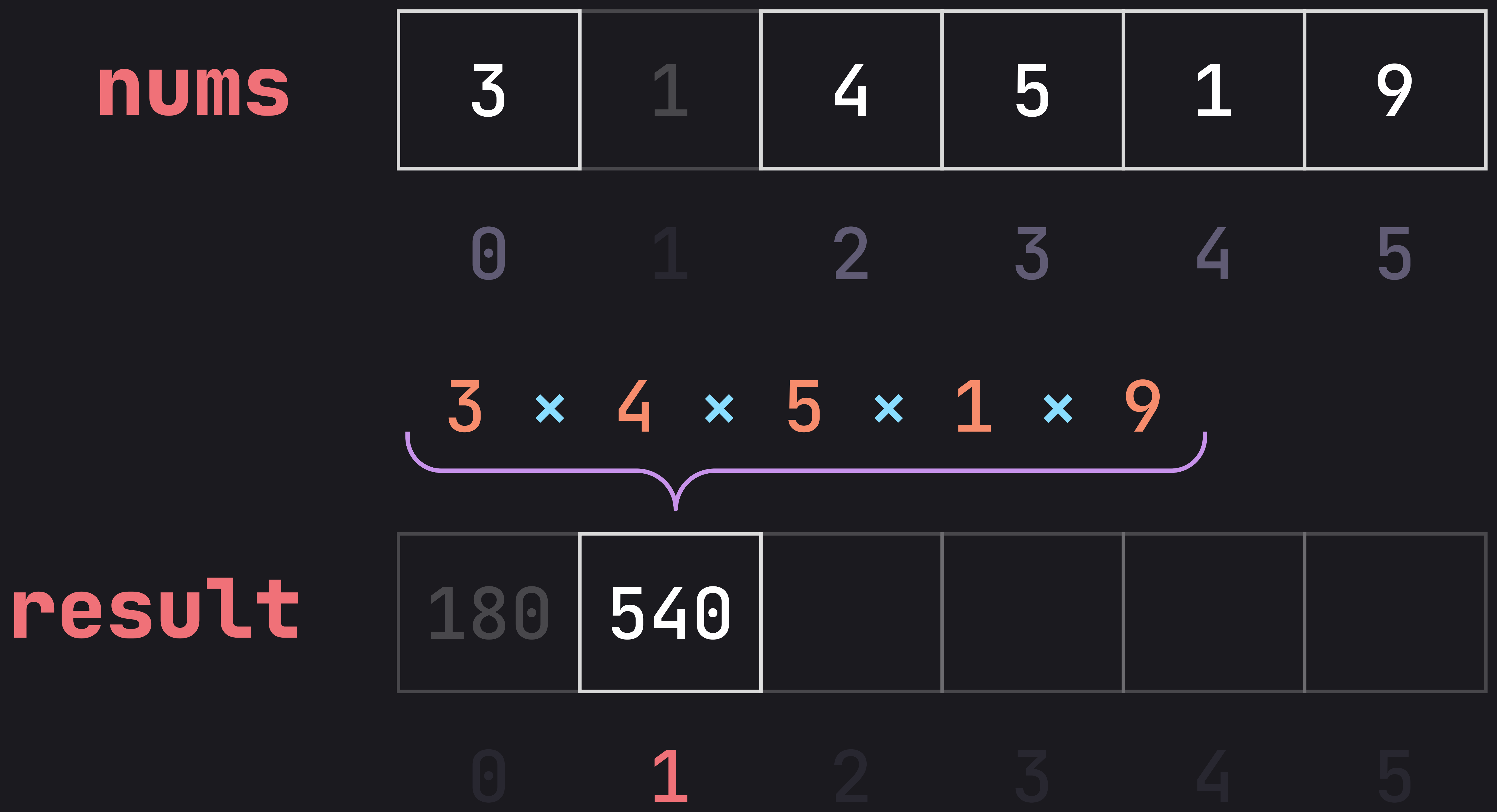
Goal: return a vector of the same size as nums such that result[i] is the product of all the integers in nums except for the ith one.

Product of Array Except Self



Goal: return a vector of the same size as nums such that result[i] is the product of all the integers in nums except for the ith one.

Product of Array Except Self

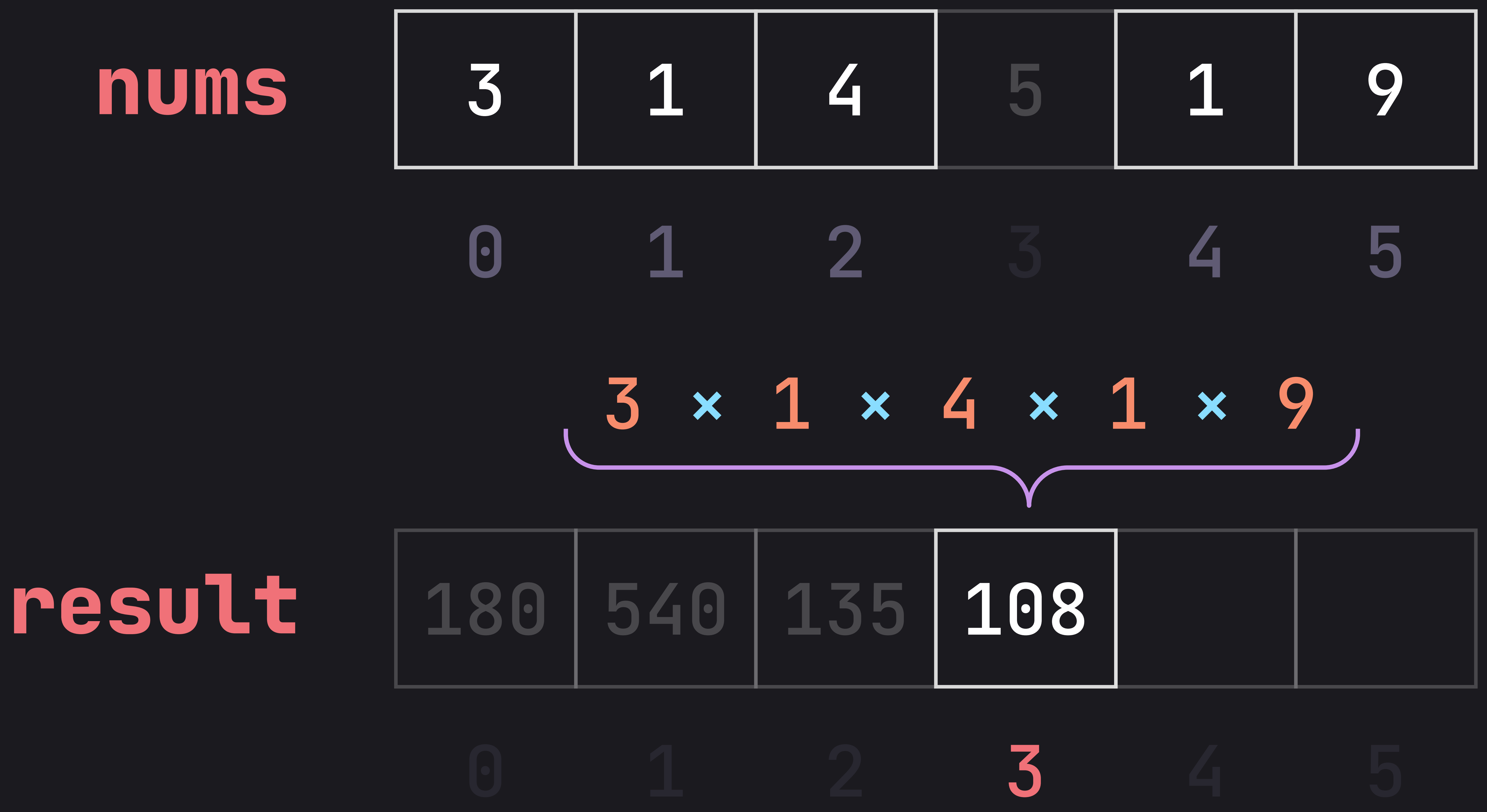


Goal: return a vector of the same size as nums such that result[i] is the product of all the integers in nums except for the ith one.

Product of Array Except Self

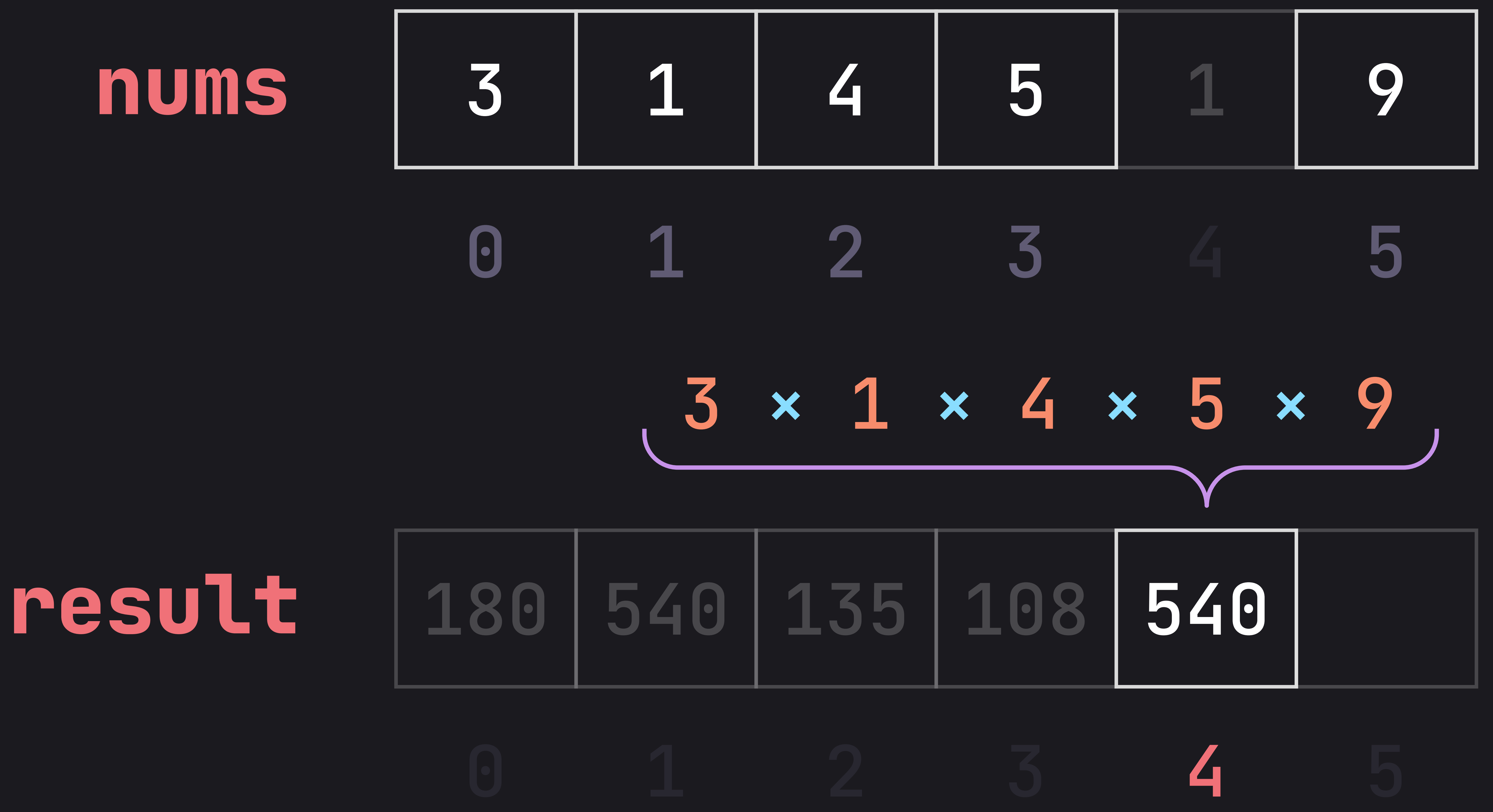
Goal: return a vector of the same size as nums such that result[i] is the product of all the integers in nums except for the ith one.

Product of Array Except Self



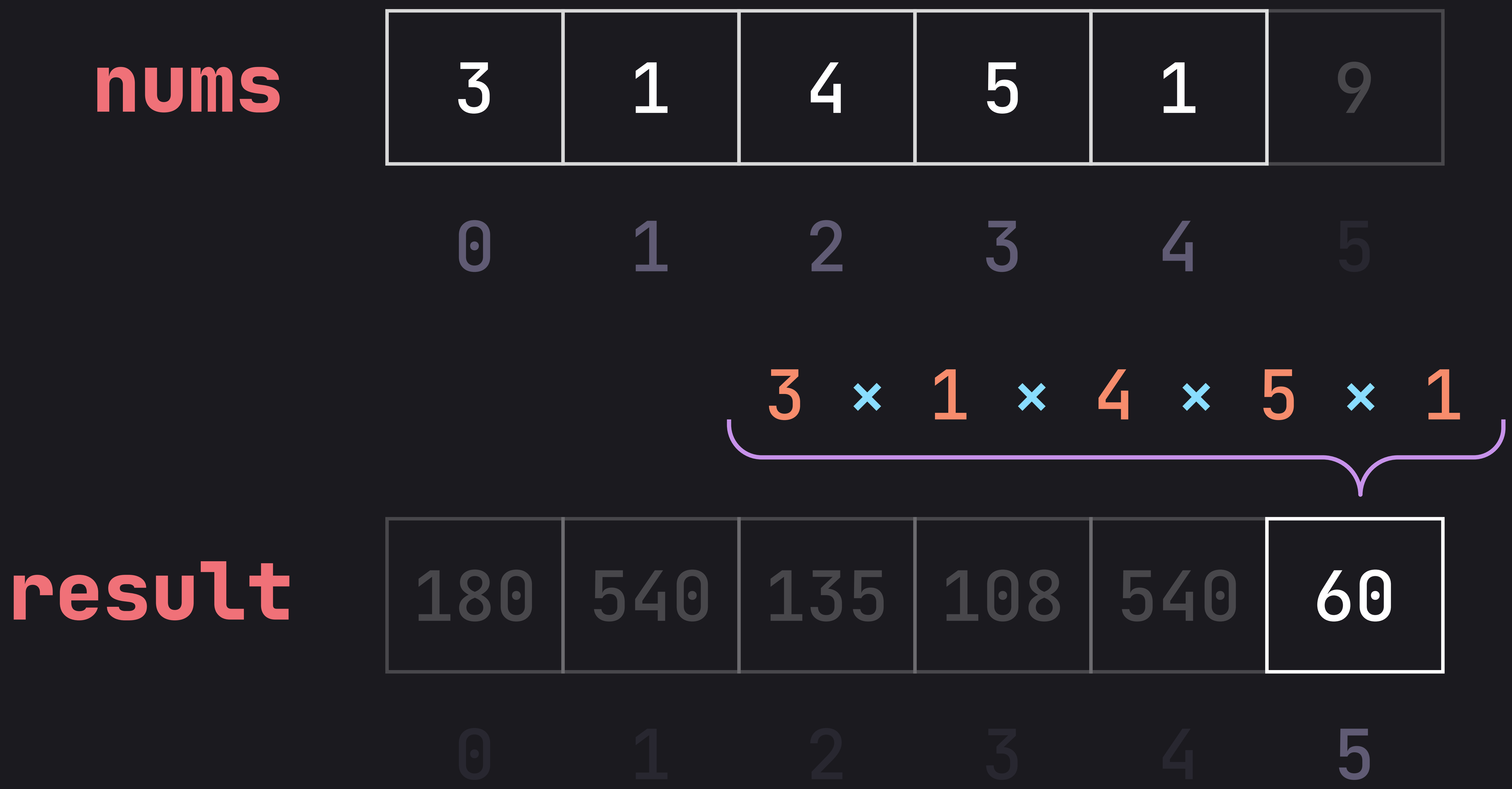
Goal: return a vector of the same size as nums such that result[i] is the product of all the integers in nums except for the ith one.

Product of Array Except Self



Goal: return a vector of the same size as nums such that result[i] is the product of all the integers in nums except for the ith one.

Product of Array Except Self



Goal: return a vector of the same size as nums such that result[i] is the product of all the integers in nums except for the ith one.

Product of Array Except Self

```cpp
std::vector<int> productExceptSelf(const std::vector<int> & nums) {
  std::vector<int> result(nums.size(), 1);
  for (std::size_t i = 0; i < result.size(); ++i) {
    for (std::size_t j = 0; j < nums.size(); ++j) {
      if (i != j) {
        result[i] *= nums[j];
      }
    }
  }
  return result;
}
```

Product of Array Except Self

```cpp
std::vector<int> productExceptSelf(const std::vector<int> & nums) {
  std::vector<int> result(nums.size(), 1);
  for (std::size_t i = 0; i < result.size(); ++i) {
    for (std::size_t j = 0; j < nums.size(); ++j) {
      if (i != j) {
        result[i] *= nums[j];
      }
    }
  }
  return result;
}
```

We start with the function definition. Notice the pass by const reference!

Product of Array Except Self

```cpp
std::vector<int> productExceptSelf(const std::vector<int> & nums) {
  std::vector<int> result(nums.size(), 1);
  for (std::size_t i = 0; i < result.size(); ++i) {
    for (std::size_t j = 0; j < nums.size(); ++j) {
      if (i != j) {
        result[i] *= nums[j];
      }
    }
  }
  return result;
}
```

Then we create a vector the same size of nums filled with 1s

(because something times 1 is itself)

Product of Array Except Self

```
std::vector<int> productExceptSelf(const std::vector<int> & nums) {
  std::vector<int> result(nums.size(), 1);
  for (std::size_t i = 0; i < result.size(); ++i) {
    for (std::size_t j = 0; j < nums.size(); ++j) {
      if (i != j) {
        result[i] *= nums[j];
      }
    }
  }
  return result;
}
```

We loop over every slot in the result

Product of Array Except Self

```cpp
std::vector<int> productExceptSelf(const std::vector<int> & nums) {
  std::vector<int> result(nums.size(), 1);
  for (std::size_t i = 0; i < result.size(); ++i) {
    for (std::size_t j = 0; j < nums.size(); ++j) {
      if (i != j) {
        result[i] *= nums[j];
      }
    }
  }
  return result;
}
```

We fill each slot with the product of all the ints in nums except the ith number

Product of Array Except Self

```cpp
std::vector<int> productExceptSelf(const std::vector<int> & nums) {
  std::vector<int> result(nums.size(), 1);
  for (std::size_t i = 0; i < result.size(); ++i) {
    for (std::size_t j = 0; j < nums.size(); ++j) {
      if (i != j) {
        result[i] *= nums[j];
      }
    }
  }
  return result;
}
```

Finally we return the array we created

Product of Array Except Self

```cpp
std::vector<int> productExceptSelf(const std::vector<int> & nums) {
  std::vector<int> result(nums.size(), 1);
  for (std::size_t i = 0; i < result.size(); ++i) {
    for (std::size_t j = 0; j < nums.size(); ++j) {
      if (i != j) {
        result[i] *= nums[j];
      }
    }
  }
  return result;
}
```

This nested loop means that we end up with a complexity of O(n^2)

Any suggestions for a more efficient algorithm? 🤔

Is there a step that we keep doing over and over which is *almost* the same each time

Goal: return a vector of the same size as nums such that result[i] is the product of all the integers in nums except for the ith one.

Any suggestions for a more efficient algorithm? 🤔

We keep **multiplying** all the numbers together!

Goal: return a vector of the same size as nums such that result[i] is the product of all the integers in nums except for the ith one.

This is...


nums
3 1 4 5 1 9
0 1 2 3 4 5
3 × 1 × 4 × 5 × 1
result
180 540 135 108 540 60
0 1 2 3 4 5

The same as this 👀

nums

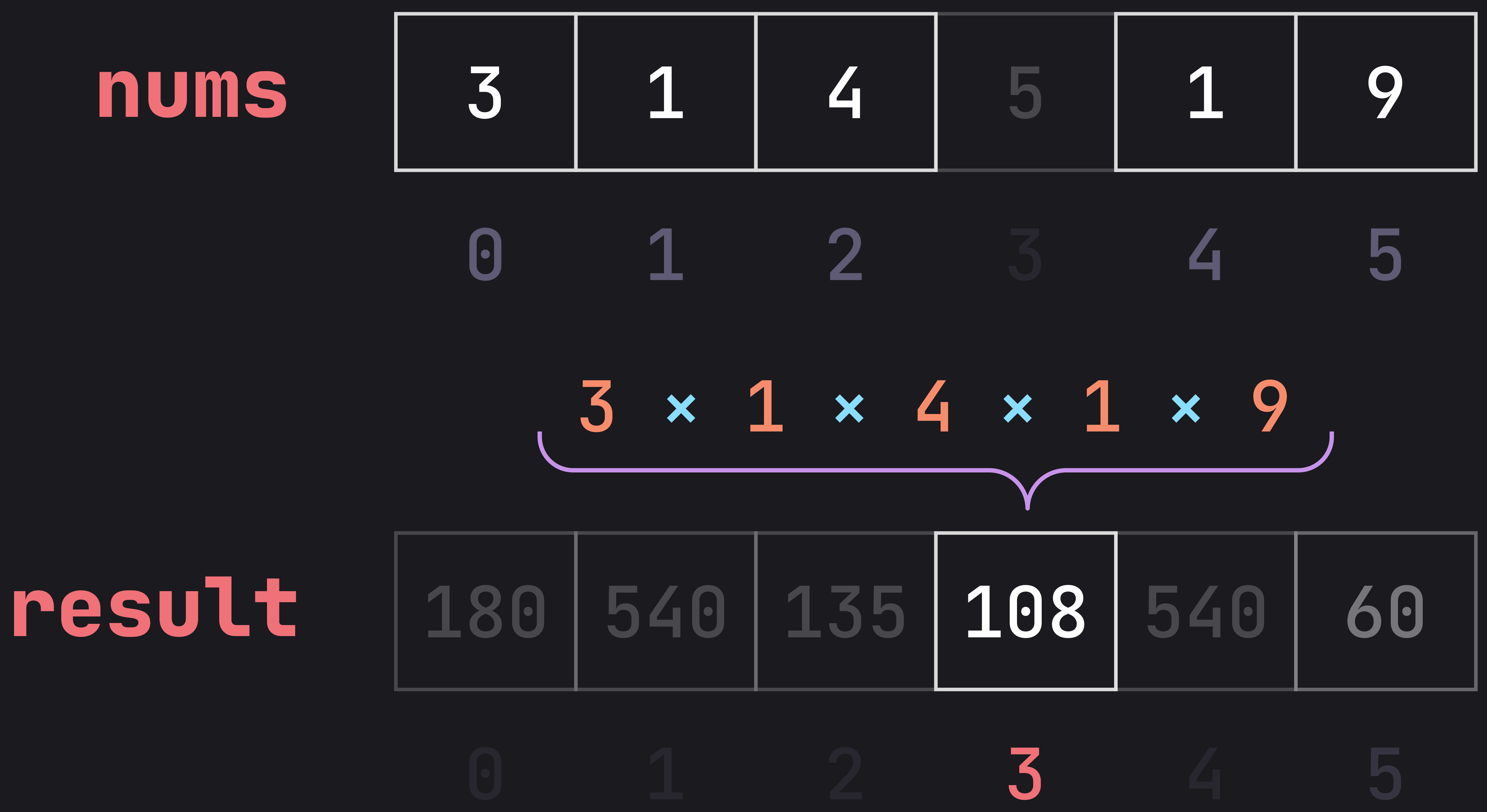
0	1	2	3	4	5
3	1	4	5	1	9

$$(3 \times 1 \times 4 \times 5 \times 1 \times 9) \div 9$$

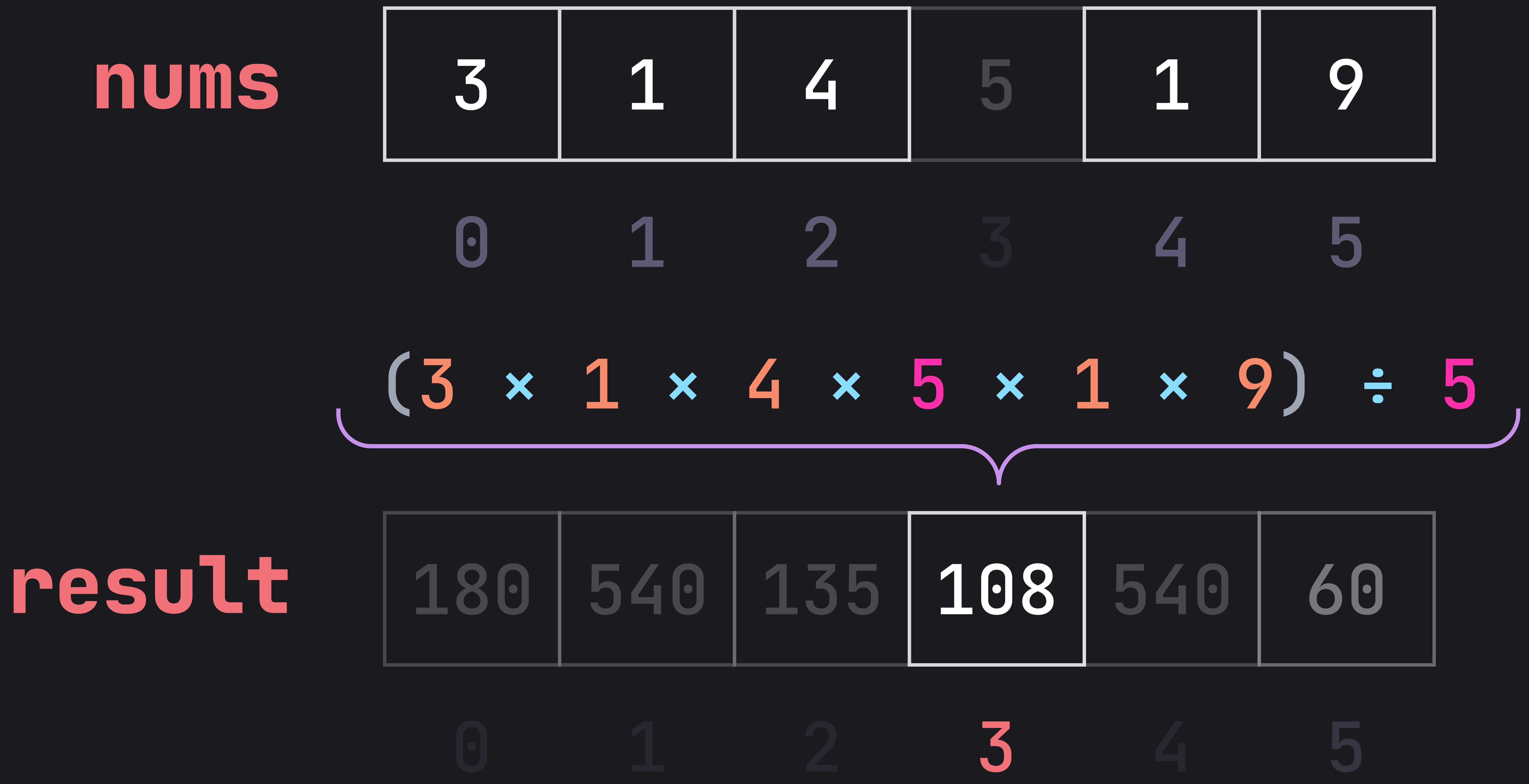
result

0	1	2	3	4	5
180	540	135	108	540	60

Just like this is...



The same as this 👀



Product of Array Except Self

```
std::vector<int> productExceptSelf(const std::vector<int> & nums) {
  std::vector<int> result(nums.size(), 0);
  int allNumsProduct = 1
  for (std::size_t i = 0; i < nums.size(); ++i) {
    allNumsProduct *= nums[i];
  }
  for (std::size_t j = 0; j < result.size(); ++j) {
    result[j] = allNumsProduct / nums[j];
  }
  return result;
}
```

Product of Array Except Self

```
std::vector<int> productExceptSelf(const std::vector<int> & nums) {
  std::vector<int> result(nums.size(), 0);
  int allNumsProduct = 1
  for (std::size_t i = 0; i < nums.size(); ++i) {
    allNumsProduct *= nums[i];
  }
  for (std::size_t j = 0; j < result.size(); ++j) {
    result[j] = allNumsProduct / nums[j];
  }
  return result;
}
```

We first calculate the product of all the ints in nums

Product of Array Except Self

```cpp
std::vector<int> productExceptSelf(const std::vector<int> & nums) {
  std::vector<int> result(nums.size(), 0);
  int allNumsProduct = 1
  for (std::size_t i = 0; i < nums.size(); ++i) {
    allNumsProduct *= nums[i];
  }
  for (std::size_t j = 0; j < result.size(); ++j) {
    result[j] = allNumsProduct / nums[j];
  }
  return result;
}
```

Then we just set each slot of the result vector to the product divided by the ith number

Product of Array Except Self

```
std::vector<int> productExceptSelf(const std::vector<int> & nums) {
  std::vector<int> result(nums.size(), 0);
  int allNumsProduct = 1
  for (std::size_t i = 0; i < nums.size(); ++i) {
    allNumsProduct *= nums[i];
  }
  for (std::size_t j = 0; j < result.size(); ++j) {
    result[j] = allNumsProduct / nums[j];
  }
  return result;
}
```

Now we are only looping through the list twice (no nested loops). So we have a complexity of O(n)

BUT!

We have a problem 👁👄👁

What happens if we have **zeros**
in our nums vector?

Dividing by zero

It should be...

But notice if we have one 0...

nums	3	1	4	5	1	0
	0	1	2	3	4	5

all of the other slots are 0

result	0	0	0	0	0	60
	0	1	2	3	4	5

And if we have two or more 0s

nums	3	0	4	5	1	0
	0	1	2	3	4	5

all of the slots are 0

result	0	0	0	0	0	0
	0	1	2	3	4	5

So we have three cases

Case 1: nums contains no 0s

```
result[i] = allNumsProduct / nums[i]
```

Case 2: nums contains one 0

```
if (nums[i] = 0)    result[i] = allNonZeroNumsProduct
            else    result[i] = 0
```

Case 3: nums contains two or more 0s

```
result[i] = 0
```

Lets code it up on Ed!