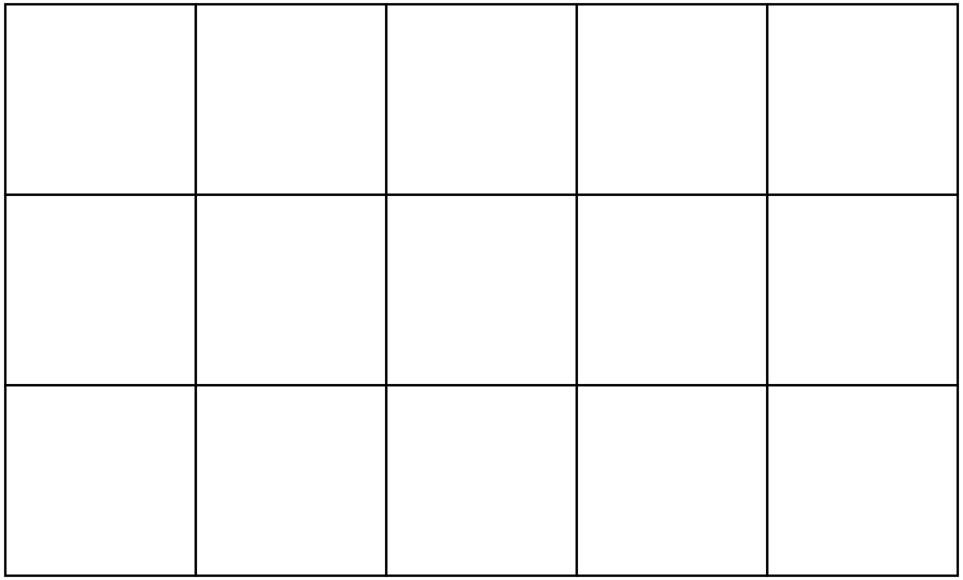
Module 2.2 - Tensor Functions

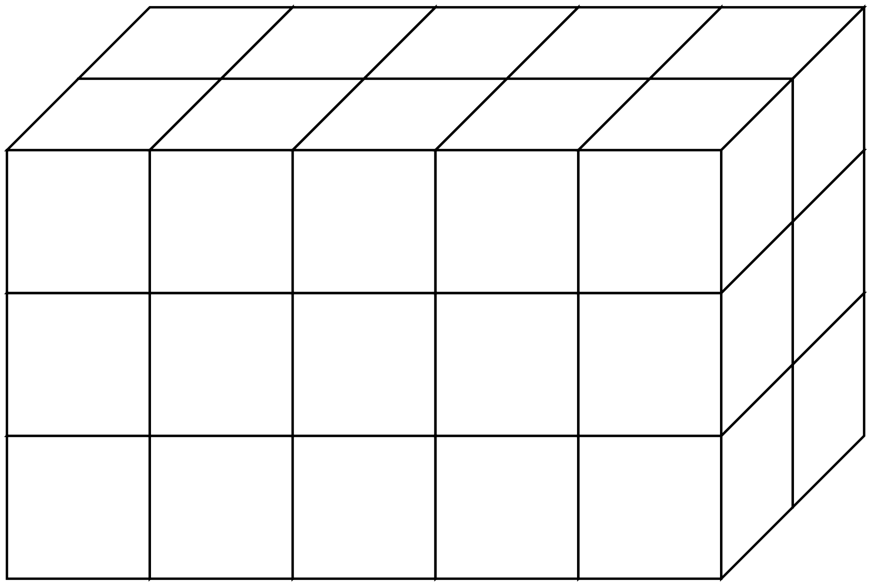
Terminology

- 2-Dimensional
- Math: Matrix



Terminology

- Arbitrary dimensions - Tensor (Array in numpy)



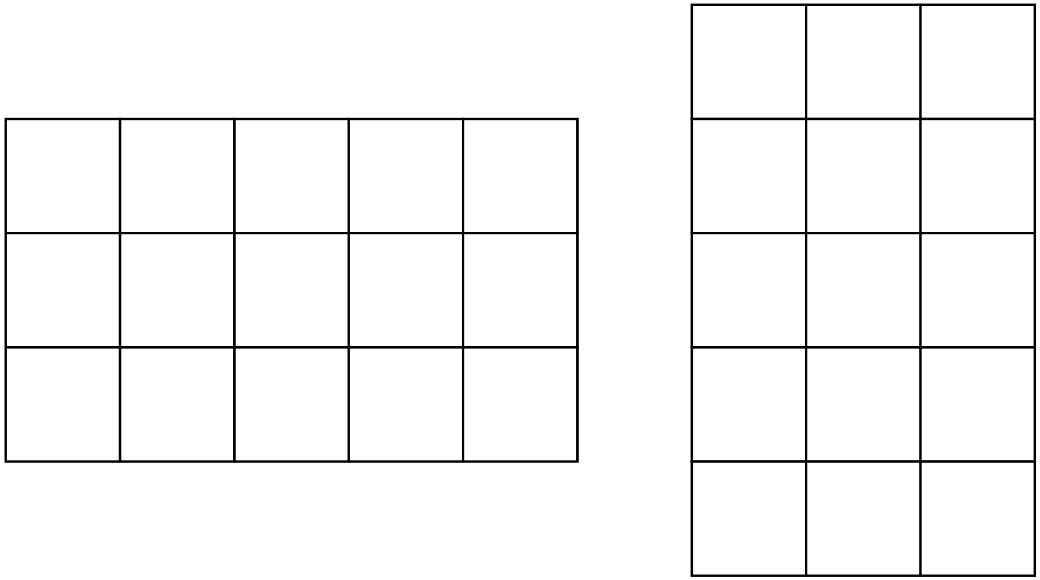
Terminology

- Dims - # dimensions (`x.dims`)
- Shape - # cells per dimension (`x.shape`)
- Size - # cells (`x.size`)

Why not just use lists?

- Functions to manipulate shape
- Mathematical notation
- Can act as Variables / Parameters
- Efficient control of memory (Module-3)

Shape - Transpose



Shape Permutation

```
x = minitorch.tensor([[1, 2, 3], [3, 2, 1]])
x.shape
```

```
(2, 3)
```

```
x.permute(1, 0).shape
```

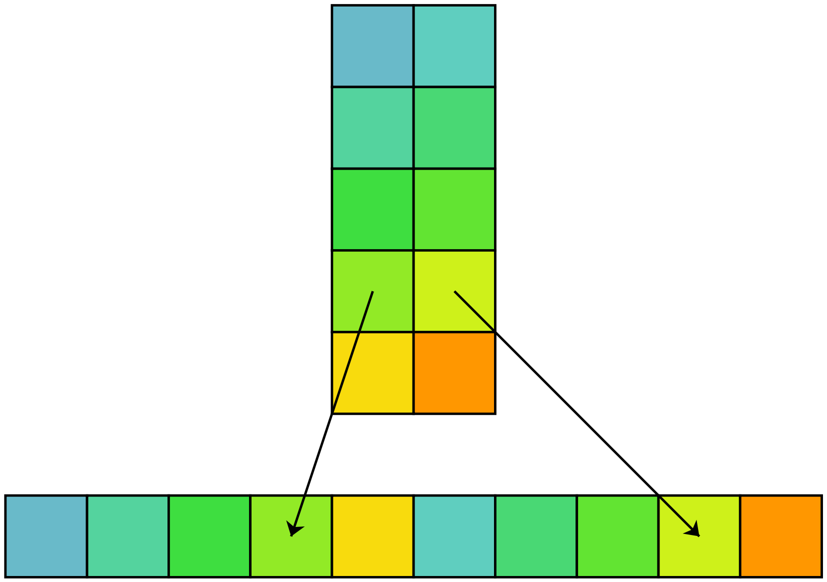
```
(3, 2)
```

Lecture Quiz 1

How does this work

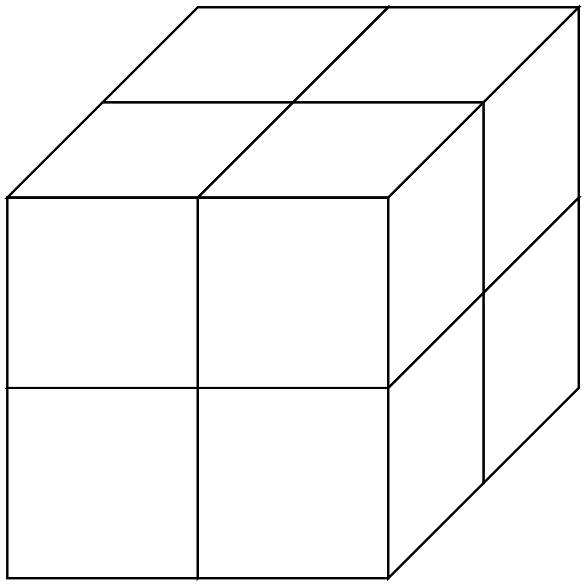
- **Storage** : 1-D array of numbers of length `size`
- **Strides** : tuple that provides the mapping from user `indexing` to the `position` in the 1-D `storage`.

Strides



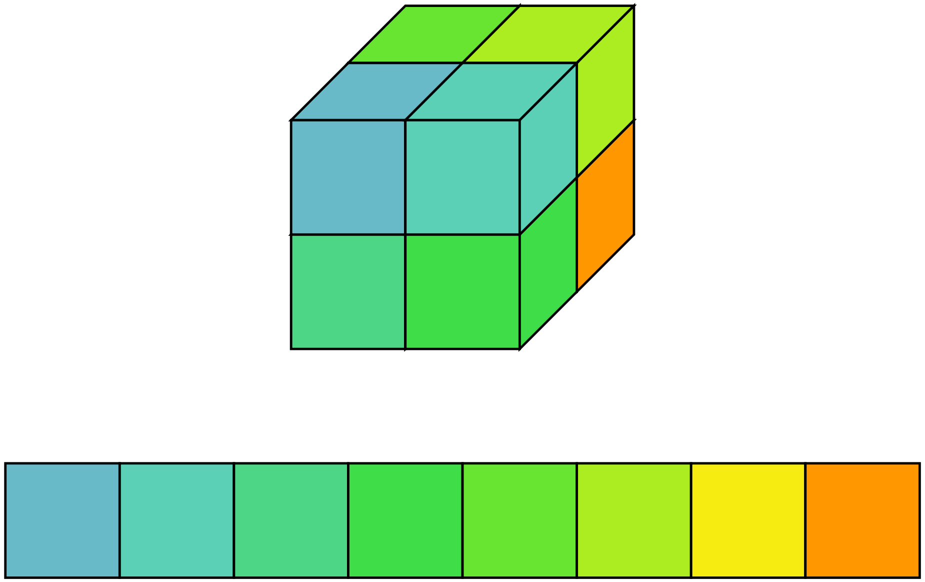
Stride Intuition

- Numerical bases,
- Index for position 0? Position 1? Position 2?



Stride Intuition

- Index for position 0? Position 1? Position 2?
- $[0,0,0],[0,0,1],[0,1,0]$



Lecture Quiz 2

Outline

- Tensor Functions
- Operations
- Broadcasting

Tensor Functions

Goal

- Support user api
- Keep track of tensor properties
- Setup fast / simple functions

Functions

- Moving from Scalar to Tensor Functions

- Implementation?

```python
def add2(a, b):
    out_tensor = minitorch.zeros(*a.shape)
    for i in range(a.shape[0]):
        for j in range(a.shape[1]):
            out_tensor[i, j] = a[i, j] + b[i, j]
    return out_tensor
```

Issues

- Different code per different dims
- Big autodiff graph
- Slow, lots of Python loops
- Lots of code

Tensor Functions

- Track graph at tensor level
- Functions wrap / unwrap Tensors

```
a = minitorch.tensor([3, 2, 1])
b = minitorch.tensor([1, 2, 3])
out = a + b
print(out)
```

```
[4.00 4.00 4.00]
```

Implementation

- `Function` class (forward / backward)
- Similar API as scalars
- Take / return Tensor objects

Operations

Implementing Tensor Functions

- Option: code `for` loop for each
- Lazy. We did this already...
- Optimization. How do we make it fast?

Strategy

- Implement high-level functions
- Lift scalar operators to tensors
- Go back and optimize high-level functions
- Customize important Functions

Tensor Functions

```
# Unary

new_tensor = a.log()

# Binary (for now, only same shape)

new_tensor = a + b

# Reductions

new_tensor = a.sum()
```

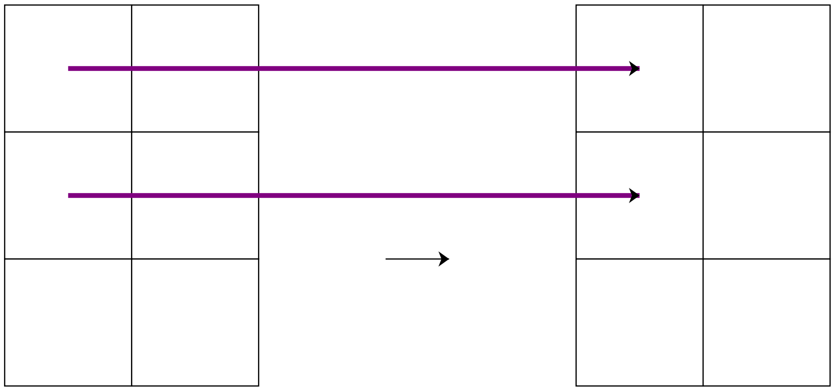
Tensor Ops

1. **Map** - Apply to all elements

2. **Zip** - Apply to all pairs

3. **Reduce** - Reduce a subset

Map

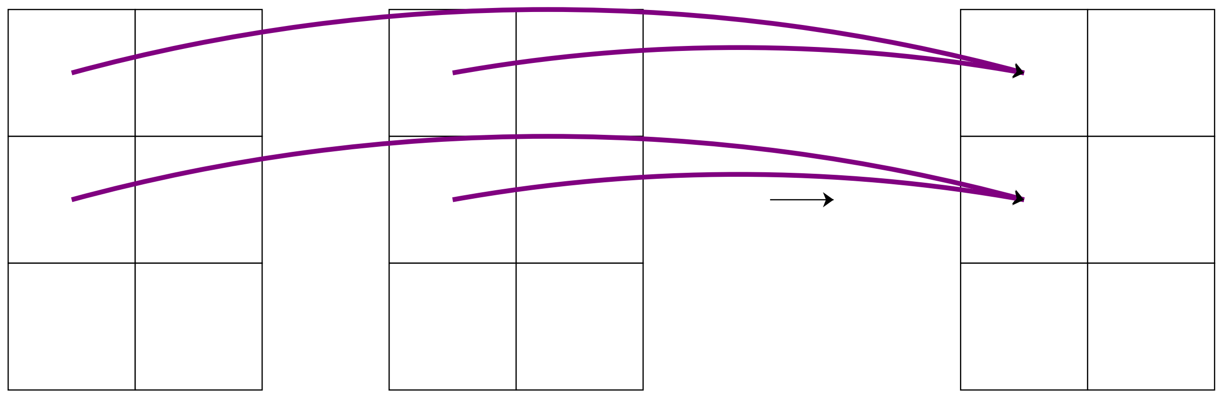


Examples: Map

Binary operations

```
new_tensor = a.log()
new_tensor = a.exp()
new_tensor = -b
```

Zip

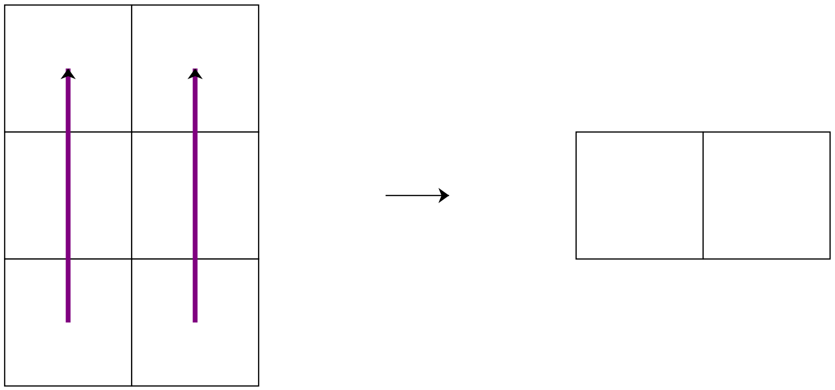


Examples: Zip

Binary operations

```
new_tensor = a + b
new_tensor = a * b
new_tensor = a < b
```

Reduce



Reduce Options

- Can reduce full tensor
- Can also just reduce 1 dimension

```
out = minitorch.rand((3, 4, 5)).mean(1)
print(out.shape)
# (3, 1, 5)
```

```
(3, 1, 5)
```

Examples: Reduce

Binary operations

```
new_tensor = a.mean()
new_tensor = out.sum(1)
```

Reduce Example

Code

Implementation Notes

- Needs to work on any strides.
- Start from output. Where does each final value come from?
- Make sure you really understand tensor data first.

Broadcasting

High Level

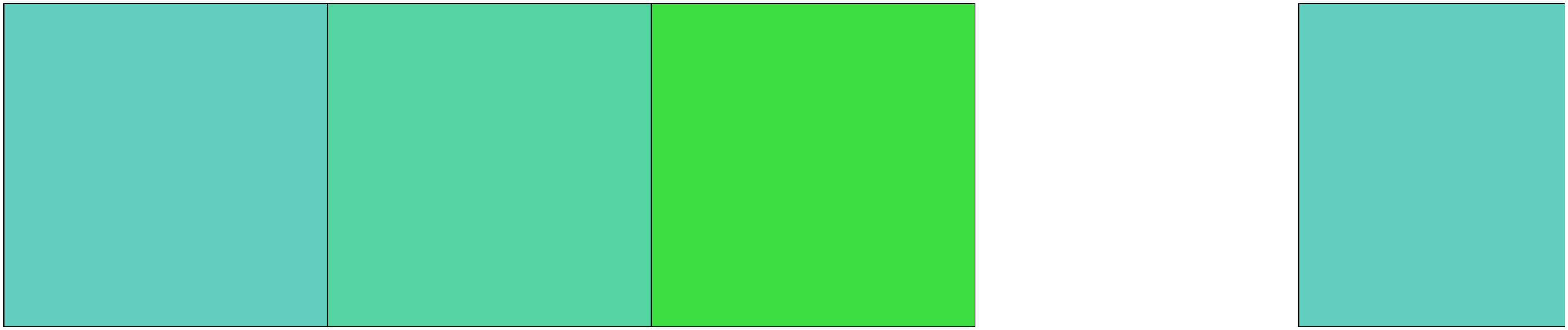
- Apply same operation multiple times
- Avoid loops and writes
- Save memory

First Challenge

- Relaxing Zip constraints
- Apply zip without shapes being identical

Motivation: Scalar Addition

```
vector1 + 10
```



Naive Scalar Addition 1

- Repeat vector-size $vector1 + [10, 10, 10]$

```
vector1 = minitorch.tensor([1, 2, 3])
print(vector1 + minitorch.tensor([10, 10, 10]))
```

```
[11.00 12.00 13.00]
```

Naive Scalar Addition 2

- Write a `for` loop

```
temp_vector = minitorch.zeros((vector1.shape[0],))
for i in range(temp_vector.shape[0]):
    temp_vector[i] = vector1[i] + 10
```

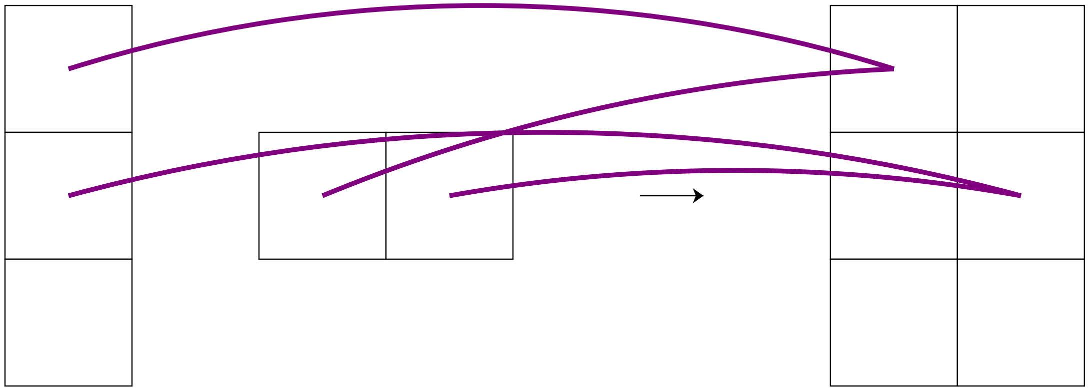
Broadcasting

- No intermediate terms
- Define rules to make different shapes work together
- Avoid for loops entirely

Zip With Broadcasting

```
a = minitorch.tensor([1, 2, 4])
b = minitorch.tensor([3, 2])
out = minitorch.zeros((3, 2))
for i in range(3):
    for j in range(2):
        out[i, j] = a[i] + b[j]
```

Zip Broadcasting



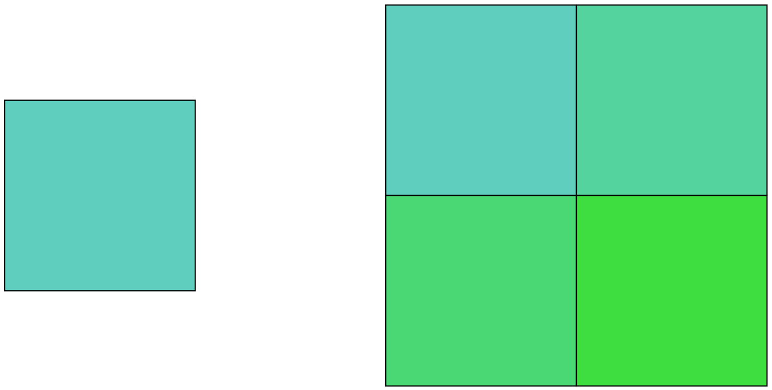
Rules

- **Rule 1**: Dimension of size 1 broadcasts with anything
- **Rule 2**: Extra dimensions of 1 can be added with `view`
- **Rule 3**: Zip automatically adds starting dims of size 1

Matrix Scalar Addition

Matrix + Scalar

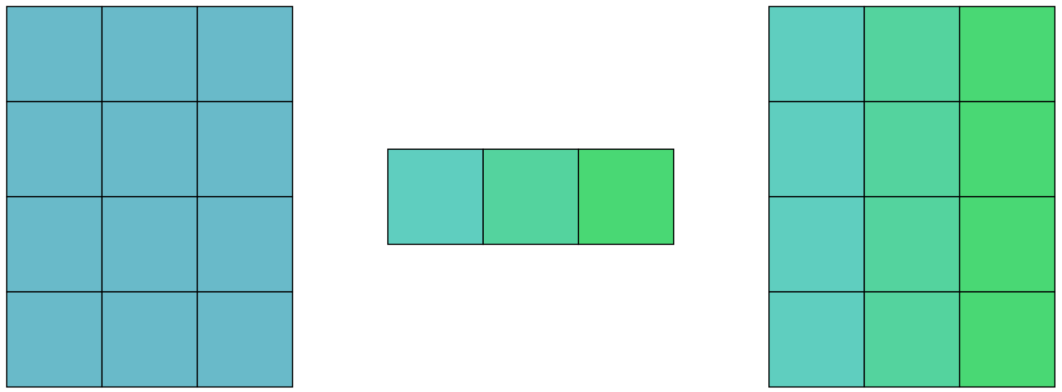
```
matrix1 + tensor([10])
```



Matrix Scalar Addition

Matrix + Vector

```
matrix1 = minitorch.zeros((4, 3))
a = matrix1.view(4, 3)
b = minitorch.tensor([1, 2, 3])
out = a + b
```



Matrix Scalar Addition

```
# Doesn't Work!
# matrix1.view(4, 3) + minitorch.tensor([1, 2, 3, 5])
```

```
# Does Work!
# matrix1.view(4, 3) + tensor([1, 2, 3, 5]).view(4, 1)
```

Applying the Rules

A	B	=
(3, 4, 5)	(3, 1, 5)	(3, 4, 5)
(3, 4, 1)	(3, 1, 5)	(3, 4, 5)
(3, 4, 1)	(1, 5)	(3, 4, 5)
(3, 4, 1)	(3, 5)	Fail

Exercises

A	B	=
(1, 3, 4)	(1, 3, 1)	
(1, 4, 4)	(3, 1, 5)	
(3, 4, 1)	(1,)	

Q&A