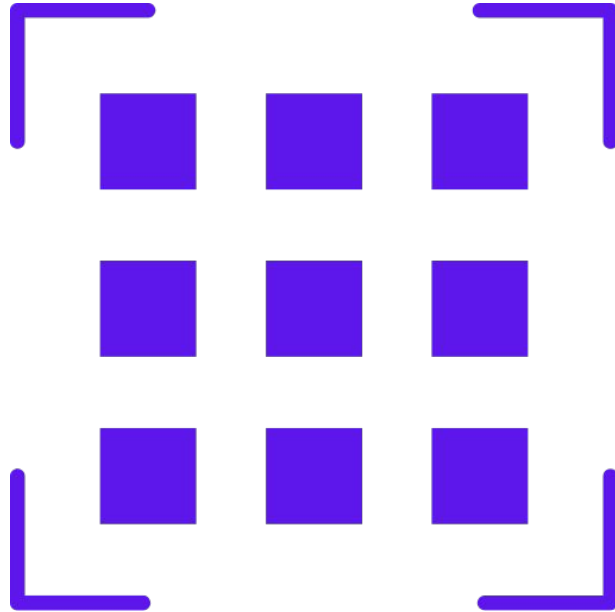
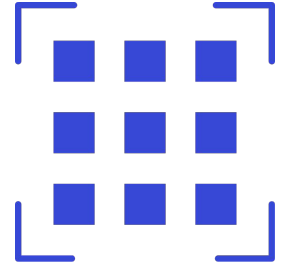


# Array/Lists And Matrices



# Pre-requisites

- Basic Python Knowledge
- Time and Space Complexity
- Willingness to learn





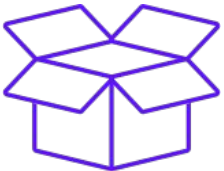
# What are Data Structures?





# Data Structures

- Particular way of **organizing**, and **storing** the data
- Tools to build efficient algorithms
- The right one depends on the task
- Common Data Structures:
  - Arrays
  - Linked Lists
  - Stacks
  - Queues
  - Trees
  - Maps(dictionaries)
  - Sets
  - Tries, etc...



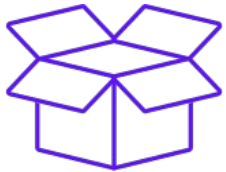
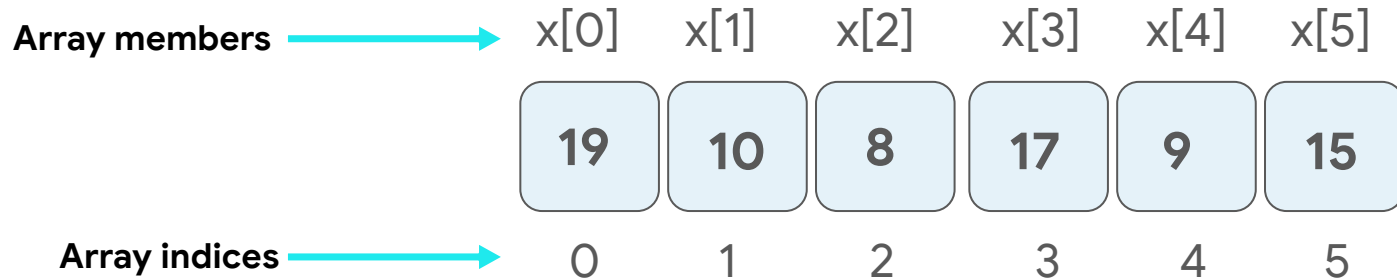
# Arrays



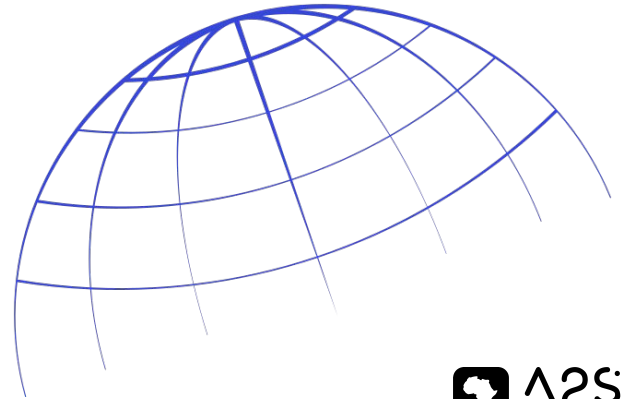
# Arrays



- Collection of items stored at **contiguous** memory locations.
- Storing multiple items of the **same type** together.
- Easier to calculate the position of each element by simply adding an offset to a base value.



# Static Arrays

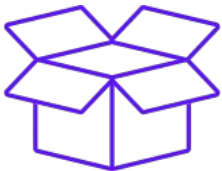




# Static Arrays

- A type of array in which the size or length is determined when the array is created and/or allocated.

	7	3	6	
	*0	*4	*8	





# Static Arrays



Value

Address

	1	3	5	
	*0	*4	*8	

Value

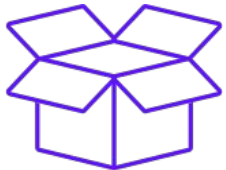
Address

	a	b	c	
	*0	*1	*2	

# Static Arrays - Reading

- Reading is Very fast :  $O(1)$
- Why?

```
print(my_array[4])
```

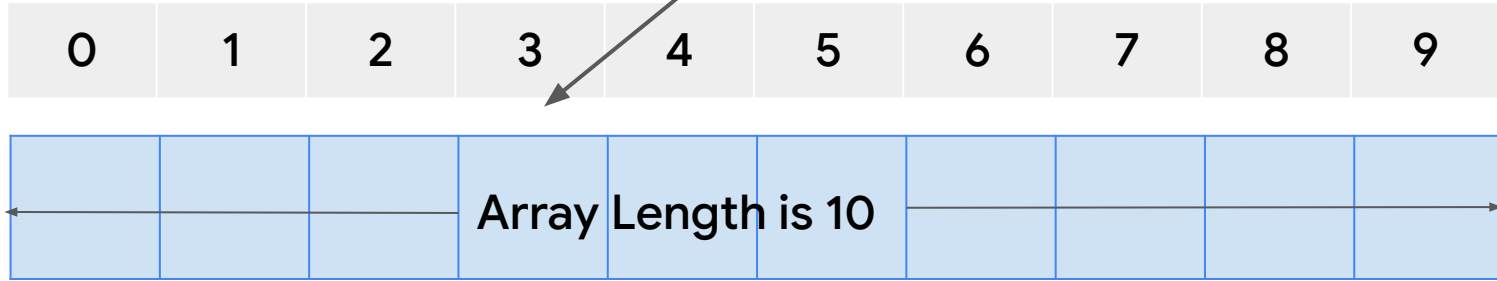




# Static Arrays - Reading

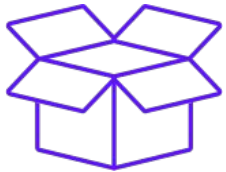
First Index

4th element



Each integer occupies 4 bytes, so let us say the 0th index is at memory location 1400

$$1400 + (3 * 4) = 1412$$



# Static Arrays - Writing

- Writing is an instant operation



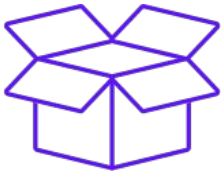
5

Value

Address

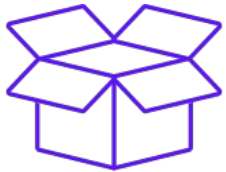
	1	3	0	
	*0	*4	*8	

5



# Static Arrays - Writing

- Overwriting is an instant operation



Value

Address

	1	3	5	
	*0	*4	*8	





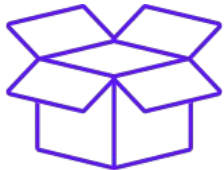
# Static Arrays - Writing

- What happens when we want to add an element to our array but don't have any empty spots?



7

Value	1	3	5	
Address	*0	*4	*8	






# Static Arrays - Writing

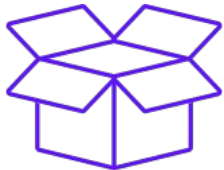
- The operating system might be using this part of the memory for some other purpose.
- So **we are not necessarily allowed** to put our seven there.



Value

Address

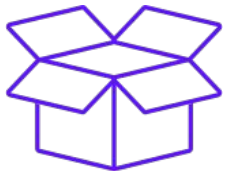
	1	3	5	
	*0	*4	*8	





# Static Arrays - Summary

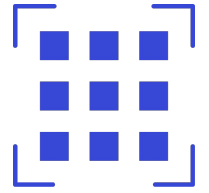
Operation	Big-O Time
Read i-th Element	$O(1)$
Write i-th Element	$O(1)$





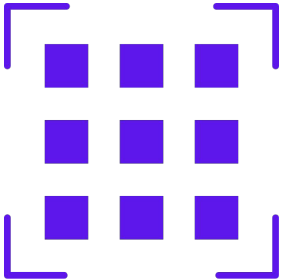
# Dynamic Arrays





# Dynamic Arrays

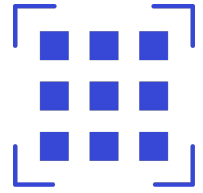
- A Dynamic array has the ability to **resize** itself automatically when an element is inserted or deleted.
- **Double the size** when it reaches capacity, **cut by half** when it's down to quarter





**Does Python list allow the storage  
of different types ?**



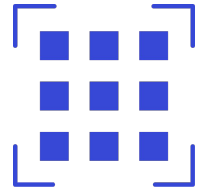


# Dynamic Arrays

- No way to find out `arr[3]` without traversing the full list

```
arr = [3, 0.5, 'a', "1"]
```

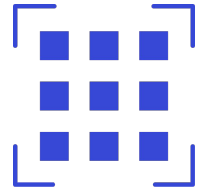
Value	3	0.5	a	"1"
Address	*0	*4	*12	*13



# Dynamic Arrays

- A list in Python is implemented as **an array of references**.
- That is that you are actually creating an array of references like so:

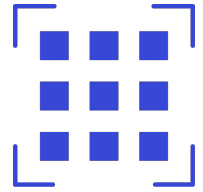
0                      1                      2                      3  
[0xa3d25342, 0x635423fa, 0xff243546, 0x2545fade]



# Dynamic Arrays

- Each element is a reference that "points" to the respective objects in memory

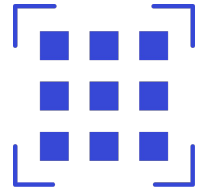
	*0	*1	*2	*3	
Reference	*10	*14	*22	*23	
Value	3	0.5	a	"1"	



# Dynamic Arrays

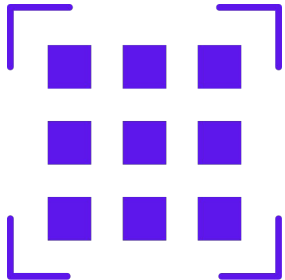
- This works because the **all references are of the same size** unlike the actual values they point to.

	<b>*0</b>	<b>*1</b>	<b>*2</b>	<b>*3</b>	
Reference	<b>*10</b>	<b>*14</b>	<b>*22</b>	<b>*23</b>	
Value	3	0.5	a	"1"	

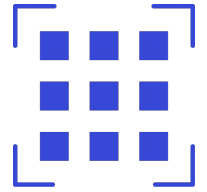


# Common Terminologies

- **Pushing/Appending** : inserting an element at the next empty spot
- **Popping**: removing the element from the end of the list

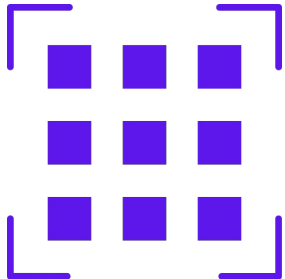






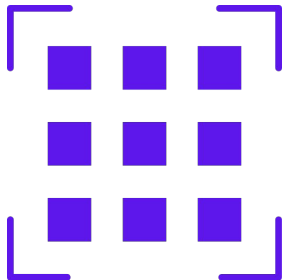
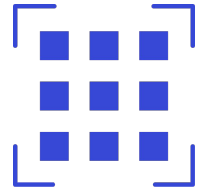
# Pushing

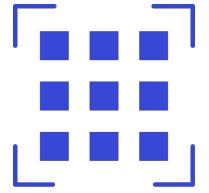
- Whenever we reach our array's capacity, our interpreter allocates another memory which is **double our original capacity** and **copies over all the old elements with now more space**.
- Isn't this operation costly?



# Pushing

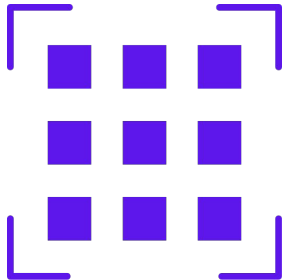
- It takes **amortized  $O(1)$  time**! Let's see why

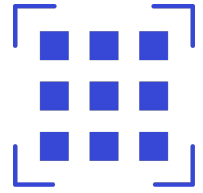




# Pushing

- Our goal is to push 8 elements to a dynamic array.
- **Assumption: Our initial capacity is of size one.**



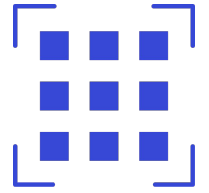


# Pushing

- After pushing 5, no place for 6



Operation count: 1

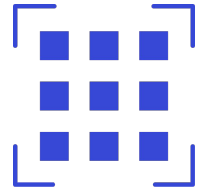


# Pushing

- Copy the original array ([5]) to the new allocated memory which is double the old size and push 6 onto it

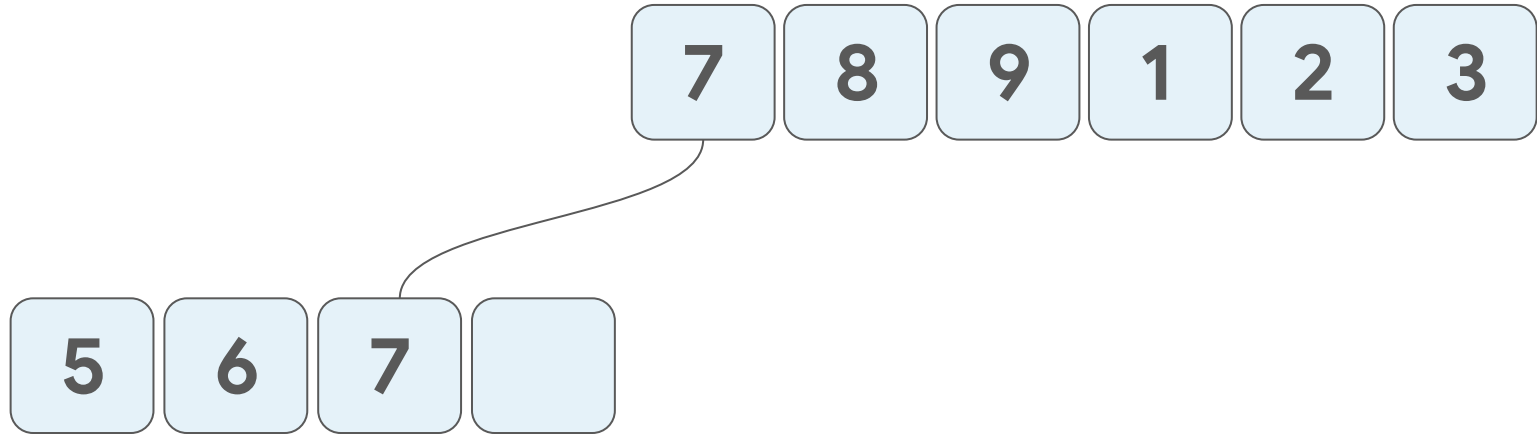


Operation count: 1 + 2

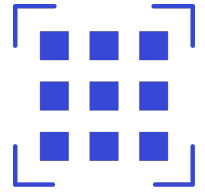


# Pushing

- Allocate a new array of size 4 and copy over all the old elements and then we append 7

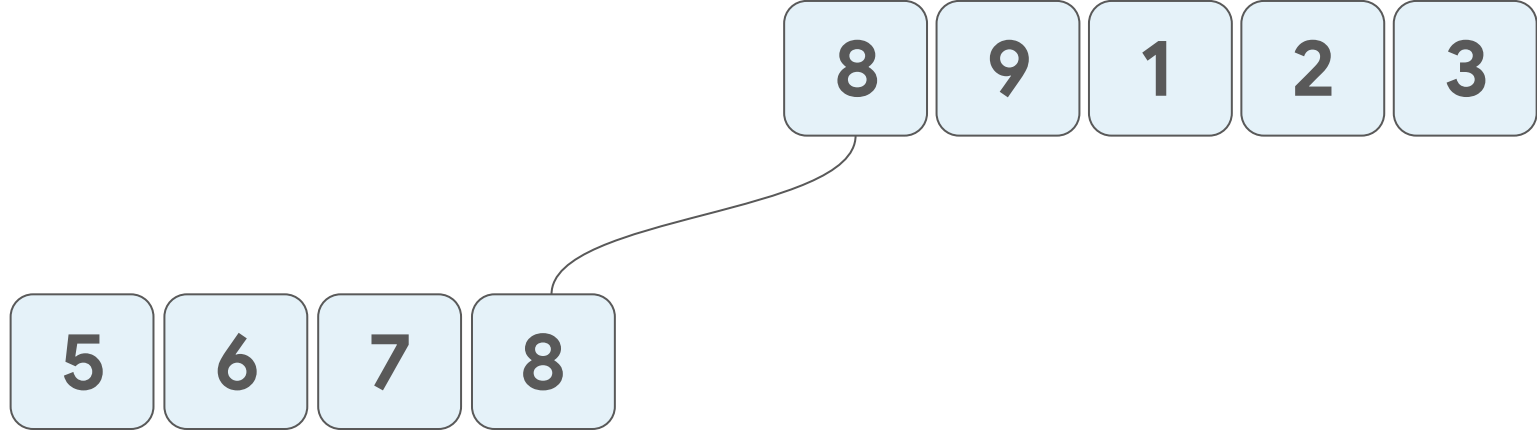


Operation count:  $1 + 2 + 3$

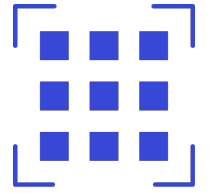


# Pushing

- Push 8 onto the array. Now we have run out of space for the next elements.

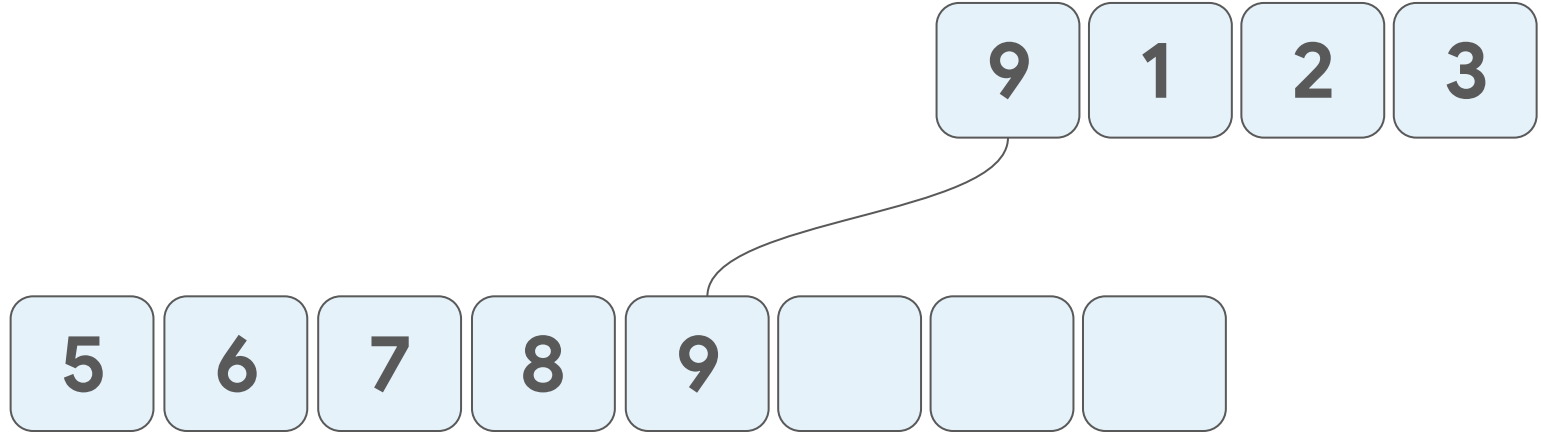


Operation count:  $1 + 2 + 4$



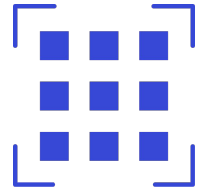
# Pushing

- Allocate a new array of size 8 and copy over all the old elements and then we append the element 9



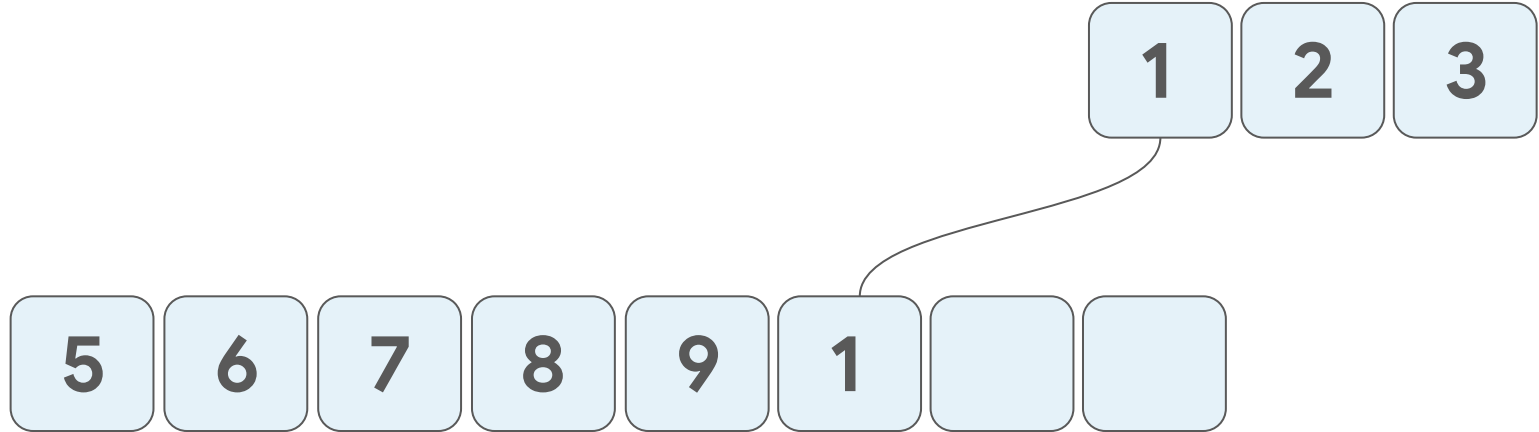
Operation count:  $1 + 2 + 4 + 5$



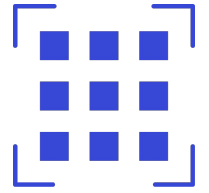


# Pushing

- Push/append 1 into the array

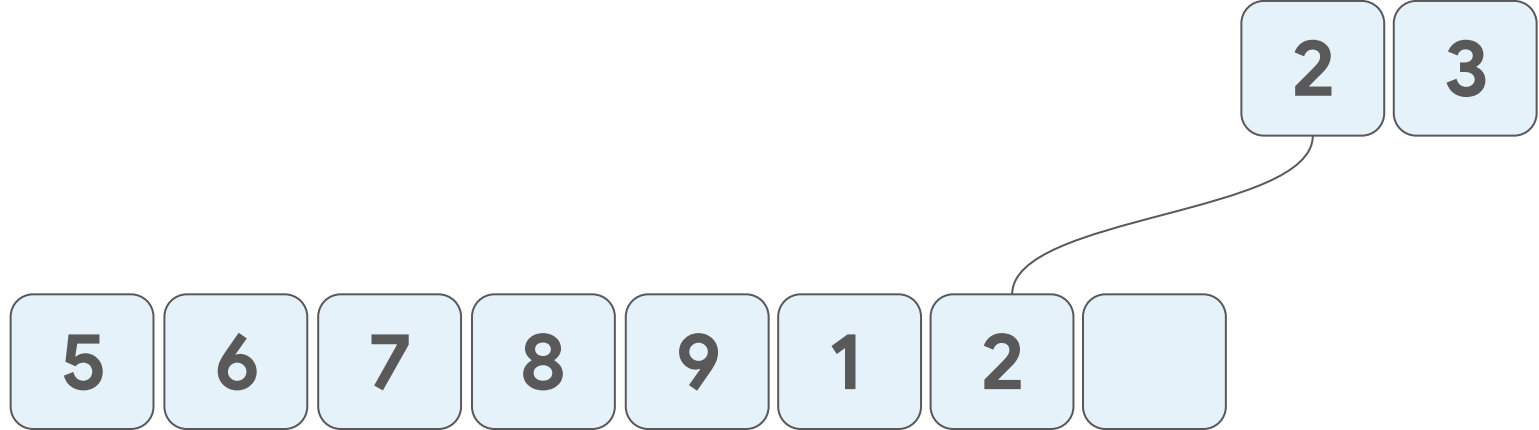


Operation count:  $1 + 2 + 4 + 6$

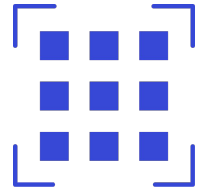


# Pushing

- Push/append 2 into the array

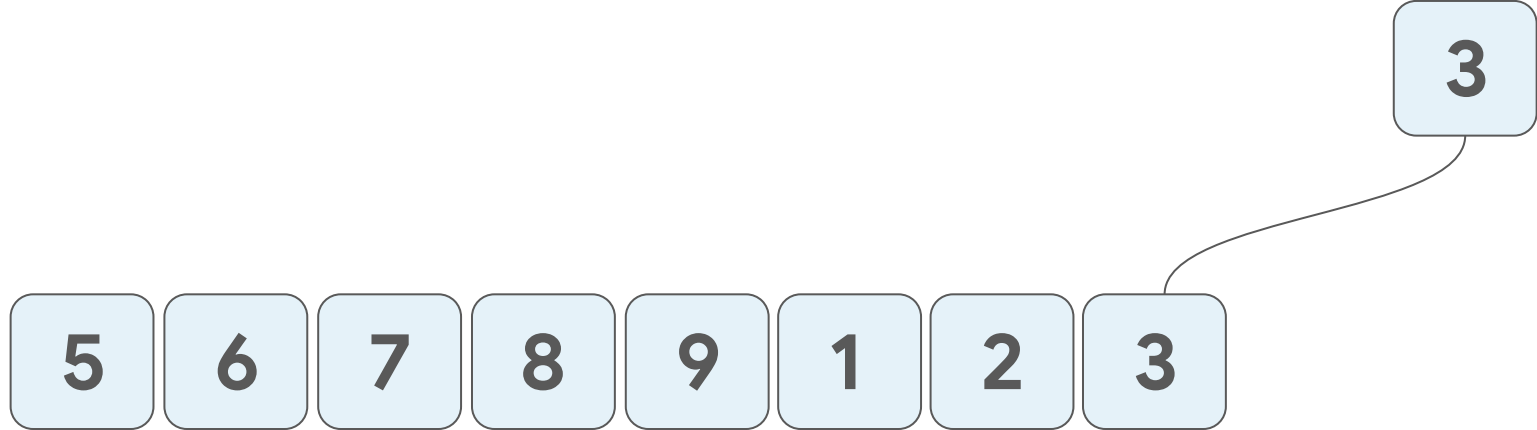


Operation count:  $1 + 2 + 4 + 7$

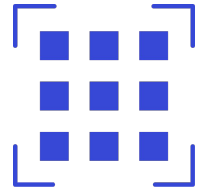


# Pushing

- Push/append 3 into the array



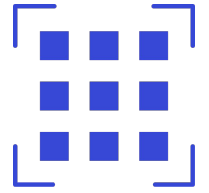
Operation count:  $1 + 2 + 4 + 8$



# Pushing- Power Series

- For a length of 8, it took us  $1 + 2 + 4 + 8$  operations.
- The last summand always dominates the sum.

$$\begin{aligned} & \boxed{\phantom{0}} \boxed{\phantom{0}} \boxed{\phantom{0}} \boxed{\phantom{0}} \boxed{\phantom{0}} \boxed{\phantom{0}} \boxed{\phantom{0}} \boxed{\phantom{0}} \\ = & \boxed{\phantom{0}} \boxed{\phantom{0}} \boxed{\phantom{0}} \boxed{\phantom{0}} + \boxed{\phantom{0}} \boxed{\phantom{0}} \boxed{\phantom{0}} \boxed{\phantom{0}} \\ = & \boxed{\phantom{0}} \boxed{\phantom{0}} \boxed{\phantom{0}} \boxed{\phantom{0}} + \boxed{\phantom{0}} \boxed{\phantom{0}} + \boxed{\phantom{0}} \boxed{\phantom{0}} \\ \geq & \boxed{\phantom{0}} \boxed{\phantom{0}} \boxed{\phantom{0}} \boxed{\phantom{0}} + \boxed{\phantom{0}} \boxed{\phantom{0}} + \boxed{\phantom{0}} + \boxed{\phantom{0}} \end{aligned}$$



# Pushing - Power Series

- For a length of 8, it took us  $1 + 2 + 4 + 8$  operations.
- The last summand always dominates the sum.

$$2 \geq 1$$

$$4 \geq 1 + 2$$

$$8 \geq 1 + 2 + 4$$

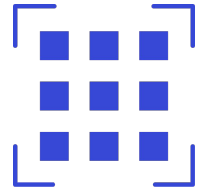
.

.

.

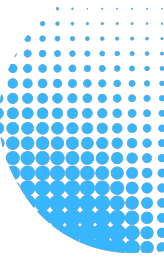
$$2^*N \geq 1 + 2 + 4 + 8 + \dots + N$$

- Thus to push  $N$  elements, it will take us no more than  $2N$  operations, making the time complexity  $O(N)$ .



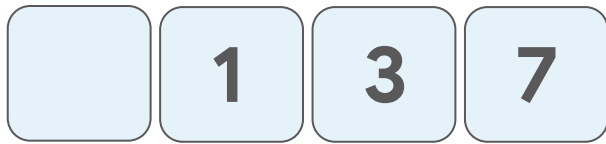
# Pushing - Power Series

- This means, to push a single element onto an array, on average, we say it is an instant -  $O(1)$  - operation.
- Because after a while, the copying operation becomes very **infrequent**.
- Hence, pushing an element onto an array is **Amortized  $O(1)$**  operation.



# Dynamic Arrays - Insertion

- Inserting an element in an arbitrary position is, in the worst case, an  $O(n)$  operation.



↑  
5

Value

Address

		1	3	7
	*0	*4	*8	*12



# Dynamic Arrays - Insertion

- We need to **shift** all elements to the right of the new element, to not lose data, and create the empty space needed.



↑  
5

Value

Address

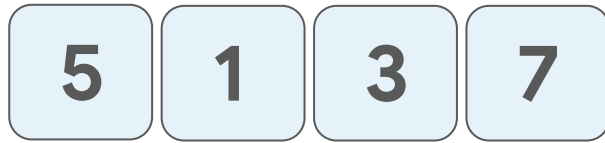
		1	3	7
	*0	*4	*8	*12





# Dynamic Arrays - Insertion

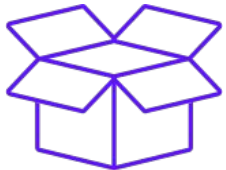
- The shifting is what costs us time.



Value

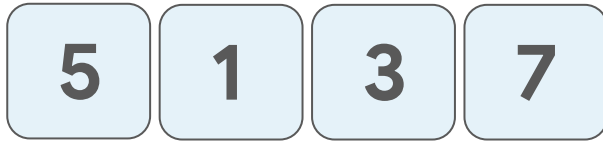
Address

	5	1	3	7
	*0	*4	*8	*12



# Dynamic Arrays - Removal

- What about removing an element?



Value

Address

	5	1	3	7
	*0	*4	*8	*12

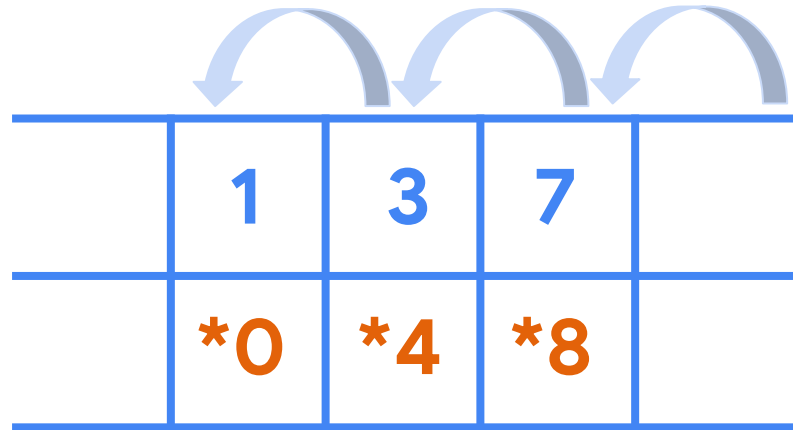
# Dynamic Arrays - Removal

- It is  $O(n)$  at worst.



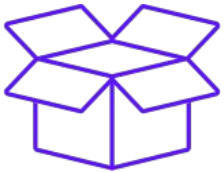
Value

Address

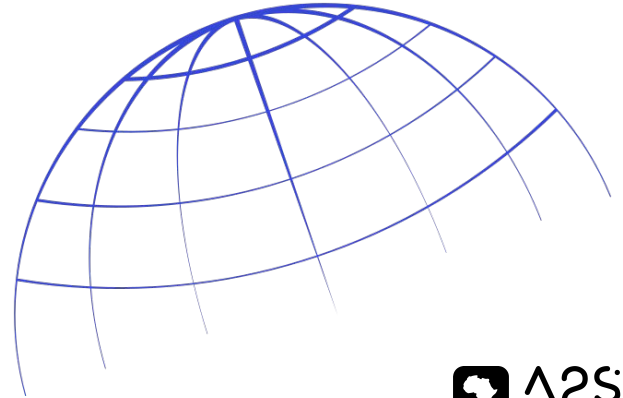


The diagram illustrates the removal of an element from a dynamic array. It shows a table with two rows: 'Value' and 'Address'. The 'Value' row contains 1, 3, and 7. The 'Address' row contains \*0, \*4, and \*8. Three curved arrows above the table indicate that elements from index 1 onwards are shifted one position to the left to fill the gap created by removing the element at index 0.

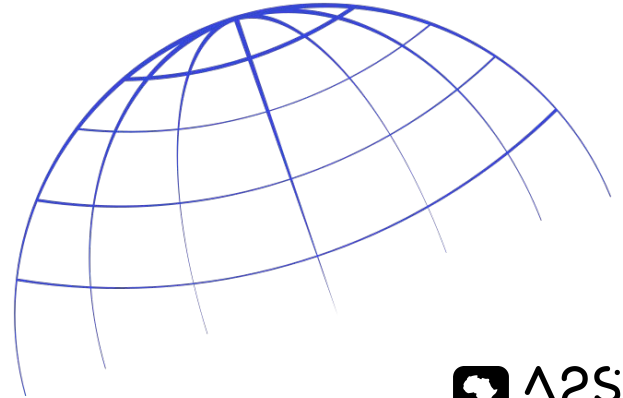
	1	3	7	
	*0	*4	*8	

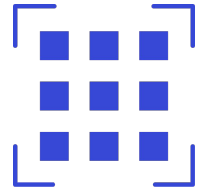


# Checkpoint



# Traversing Lists





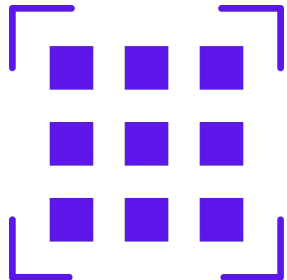
# Traversing Lists - Reverse Traversal

## Approach

- Start from the last index
- Decrement by 1 until -1

## Implementation

```
for index in range(len(array)-1, -1, -1):  
    print(array[index])
```



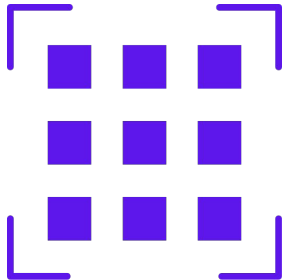
# Traversing Lists - Even/Odd Indices Traversal

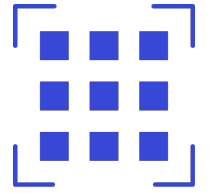
## Approach

- Start from the first even/odd index
- Increment by 2

## Implementation

```
step = 2 # incremental value
# even indices traversal
size = len(array)
for index in range(0, size, step):
    print(array[index])
```





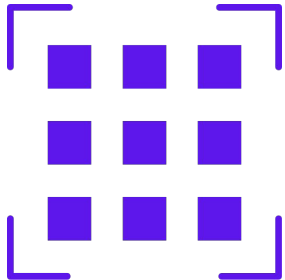
# Traversing Lists - Circular Traversal

## Approach

- Start from first index to  $\text{size} \times 2$
- Incrementally go 1 step but modulo index by size

## Implementation

```
size = len(array)
for index in range(0, 2 * size):
    value = array[index % size]
    print(value)
```



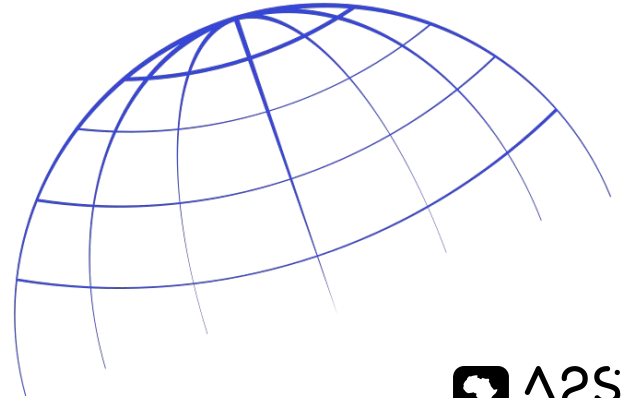


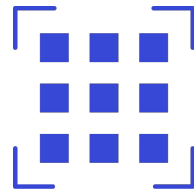
# Pair Programming

## Find the winner of the circular game



# Swapping Elements





# Swapping Elements

## Approach

- Approach 1
  - Store first element in a variable
  - Change first element with second element
  - Change second element with stored value
- Approach 2 : Use **tuple unpacking** to swap

## Implementation

```
# approach 1
```

```
temp = arr[i]
```

```
arr[i] = arr[j]
```

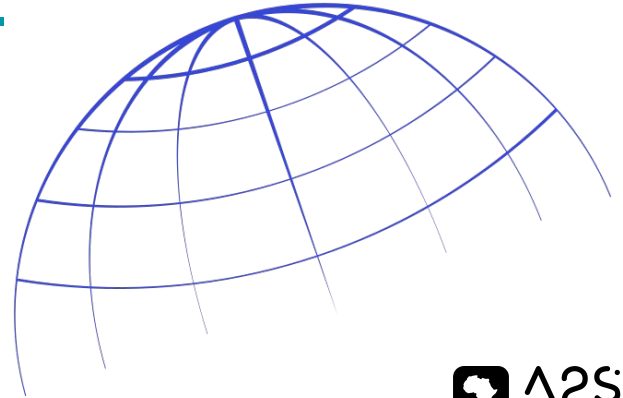
```
arr[j] = temp
```

```
# approach 2
```

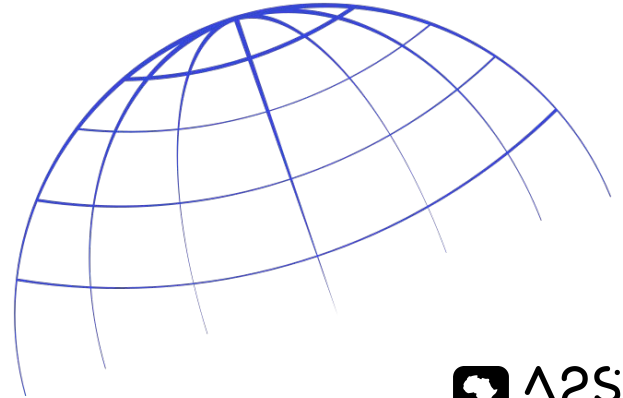
```
arr[i], arr[j] = arr[j], arr[i]
```

# Pair Programming

## Reverse string

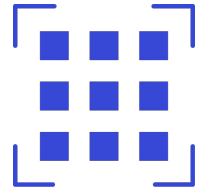


# Storing letter and Numbers



# Storing - Using Arrays as Dictionaries

- For Letters
  - Character's ASCII value can be the index by correcting the offset by 65 ('A') for uppercase letters and 97 ('a') for lower case letters
- For Numbers
  - The number itself is the key
- Note: the advantage of this approach is that you can use built in functions like `sum`, `sort`, `max`, `min` on the frequency list whereas maps don't give you this flexibilities.

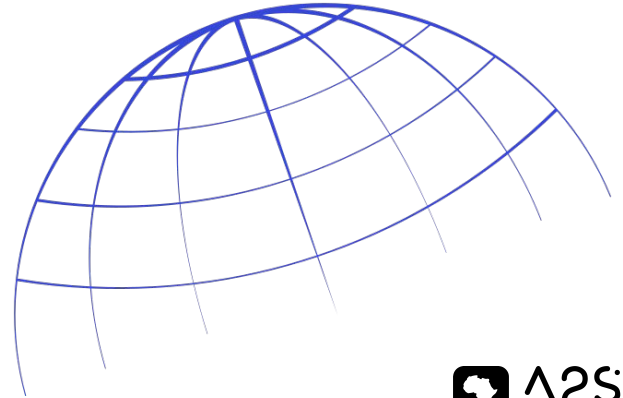


# Storing - Implementation

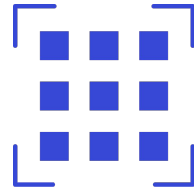
```
word = "abceabce"
arrayDictionary = [0]*26
offset = ord('a') # 'a' has ASCII value of 97
for char in word:
    ascii = ord(char)
    arrayDictionary[ascii - offset] += 1

print(arrayDictionary)
# [2, 2, 2, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
# a b c d e f g h i j k l m n, o p q r s t u v w x y z
```

# Read and Write Indices







# Read and Write and Indices

## Approach

- Set your read index accordingly to your need
- Set your write index accordingly to your need
- Your read should always move
- Your write only moves after write operation

## Implementation ([Question Link](#))

```
def moveZeroes(self, nums: List[int])  
    write = 0  
    read = 0  
  
    while read < len(nums):  
        if nums[read] != 0:  
            temp = nums[read]  
            nums[read] = nums[write]  
            nums[write] = temp  
  
            write = write + 1  
  
        read = read + 1
```

# Pair Programming

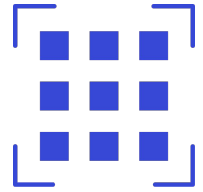
## Apply operations to an array



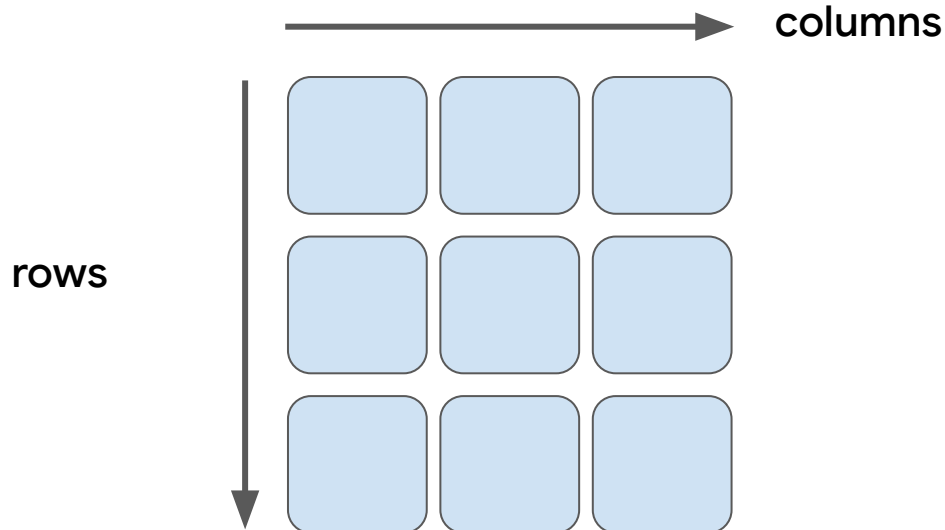
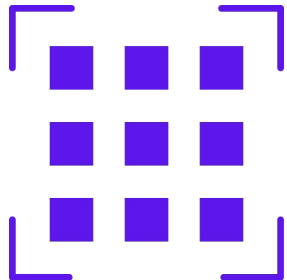
# Matrices

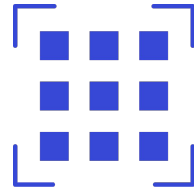


# Matrices



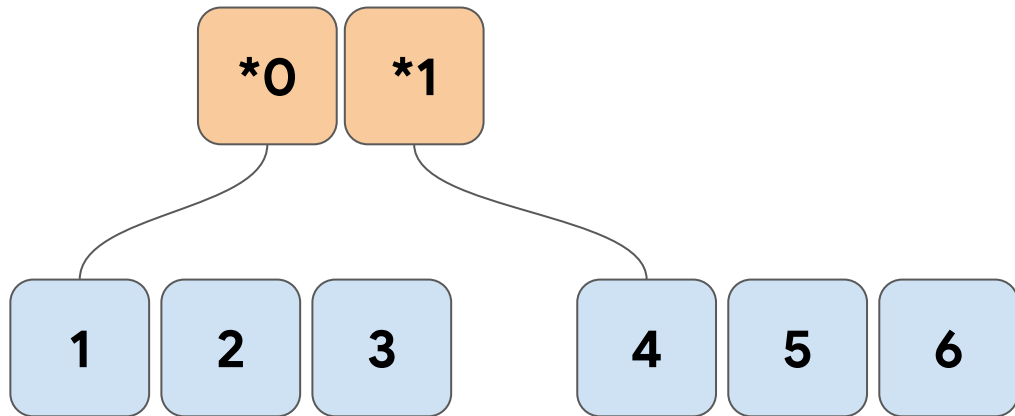
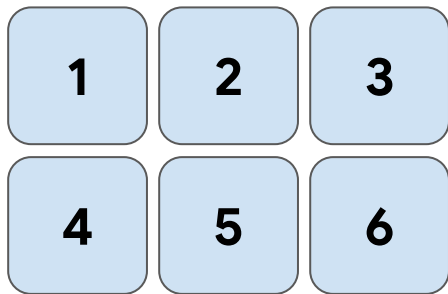
- Two-dimensional arrays can be defined as **arrays within an array**.
- **2D** arrays erected as matrices, which is a collection of rows and columns.



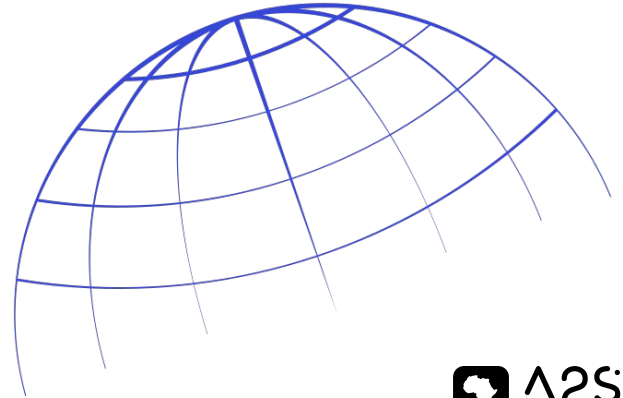


# Matrices

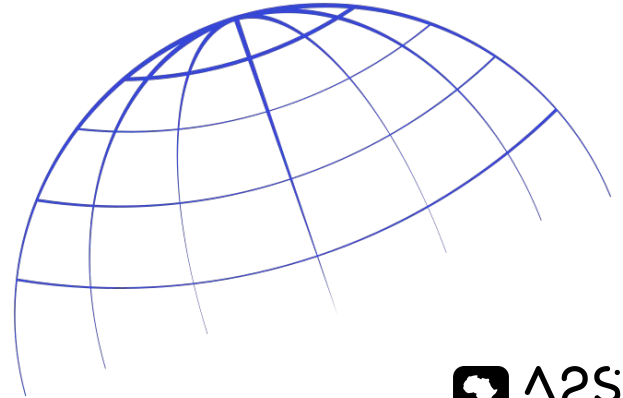
- You should think two dimensional arrays as **matrices**.
- In reality, they are arrays holding references to other arrays



# Common Approaches



# Traversing Matrices



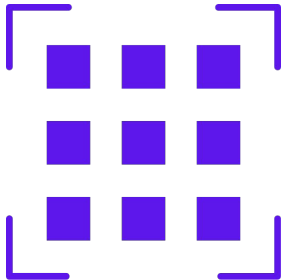
# Traversing - from Top-left to Bottom-right

## Approach

- Start from (0, 0)
- Increment column index by one for each row you visit

## Implementation

```
for row_idx in range(len(matrix)):  
    for col_idx in range(len(matrix[0])):  
        print(matrix[row_idx][col_idx])
```





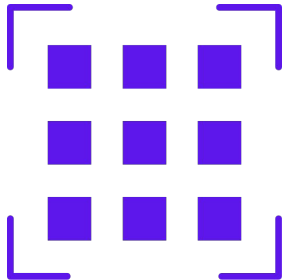
# Traversing - From Bottom-right to Top-left

## Approach

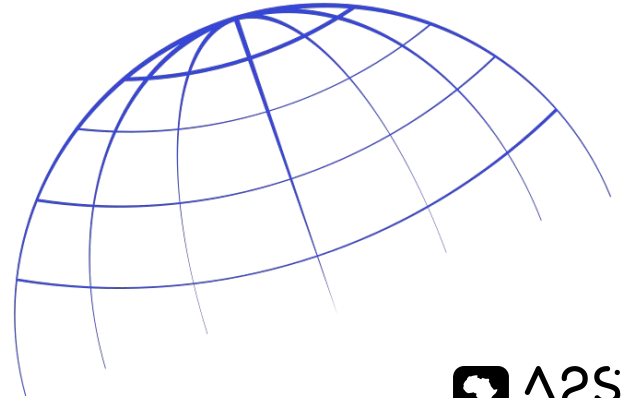
- Start from (last\_row - 1, last\_col - 1)
- Decrement column index by one for each row you visit.

## Implementation

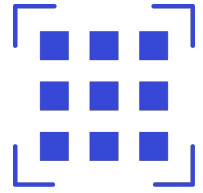
```
for row_idx in range(len(matrix) - 1, -1, -1):  
    for col_idx in range(len(matrix[0]) - 1, -1, -1):  
        print(matrix[row_idx][col_idx])
```



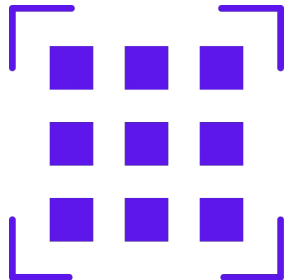
# Enumerating Cells/ 2D - 1D Mapping

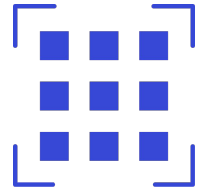


# Enumerating Cells - Approach



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

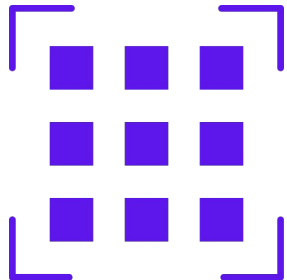


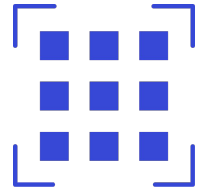


# Enumerating Cells - Approach

Consecutive elements in a column differ by **the number of columns**

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

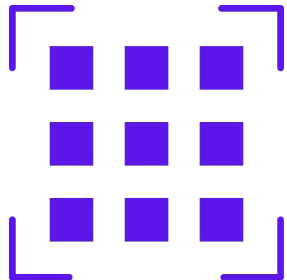




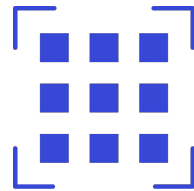
# Enumerating Cells - Approach

Consecutive elements in a row differ by 1

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

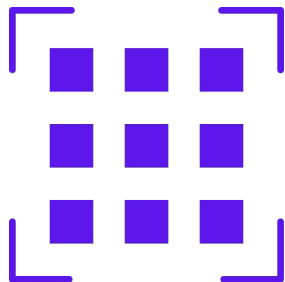


# Enumerating Cells - Approach



Therefore the 1D number of a cell is  $\text{row\_number} * \text{n\_columns} + \text{column\_number}$

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

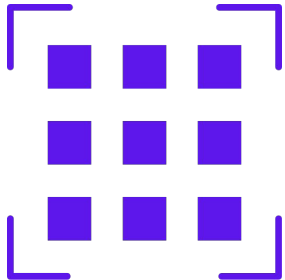


# Enumerating Cells - Remainder Theorem

```
cell_number = row_number*n_columns + column_number
```

```
row_number = cell_number // n_columns
```

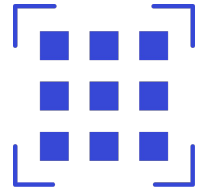
```
column_number = cell_number % n_columns
```



# Diagonal Keys



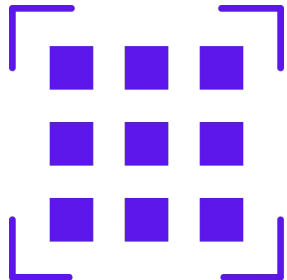


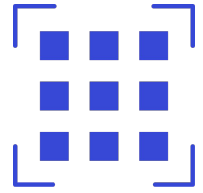


# Diagonal Keys

What is common about the highlighted column and row numbers?

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

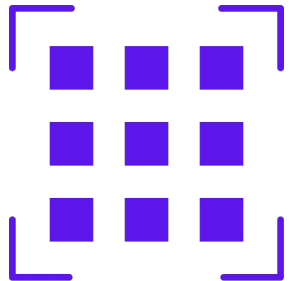




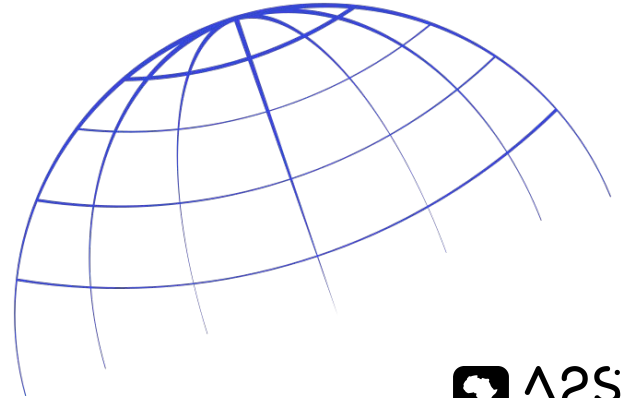
# Diagonal Keys

What is common about the highlighted column and row numbers?

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



# Common Pitfalls



# Common Pitfalls - Index Out of Bound

- Trying to access elements using indices that are **greater (or equal to)** than the size of our array will result in this exception.

Testcase Result 

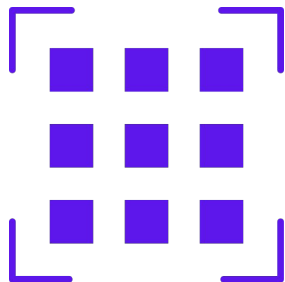
```
IndexError: list index out of range
  print(nums[i])
Line 4 in twoSum (Solution.py)
    ret = Solution().twoSum(param_1, param_2)
Line 28 in _driver (Solution.py)
    _driver()
Line 39 in <module> (Solution.py)
```

Stdout

2

Console ^

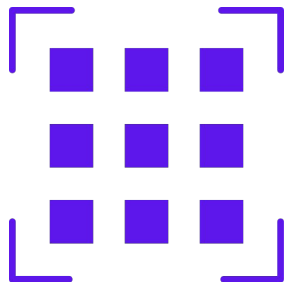
 Run Submit



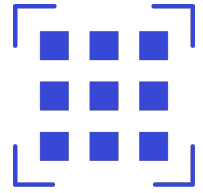
# Common Pitfalls - Negative Indexing

- Because we may use negative indices in Python, it is a typical problem to use negative indices accidentally, which can result in a highly troublesome misunderstanding if no exception is triggered.

Negative indices:	-4	-3	-2	-1
Positive indices:	0	1	2	3
Values:	[1	, 2	, 3	, 4]

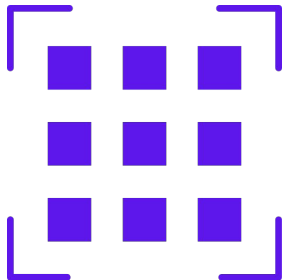


# Common Pitfalls - Copying lists



- A list is a reference. Take care when you want to pass **only the content** of the list.

```
nums[::]  
nums.copy()
```

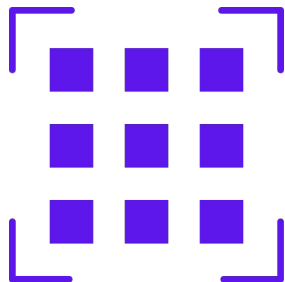


# Common Pitfalls - Initializing with \*

- The `*` operator can be used as `[object]*n` where `n` is the number of elements in the array.
- In the case of 2D arrays, this will result in **shallow lists**
- Hence using list comprehensions is a safer way to create 2D lists.

```
row = len(matrix)
col = len(matrix[0])
```

```
transposed = [[0]*rows for _ in range(cols)]
```



# Common Pitfalls - Using insert , pop and in

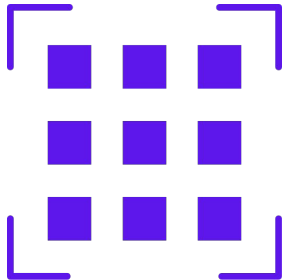
- insert - inserts a given element at a given index
  - `insert(i, value)` :  $O(N)$
- pop - returns the value at a given index and removes it from the list where the default is the last index
  - `pop` :  $O(1)$
  - `pop(i)` :  $O(n)$
- The `in` operator returns a boolean denoting whether an element is present in the iterable data structure we use it on.
- Lists  $O(N)$
- Map, Set  $O(1)$



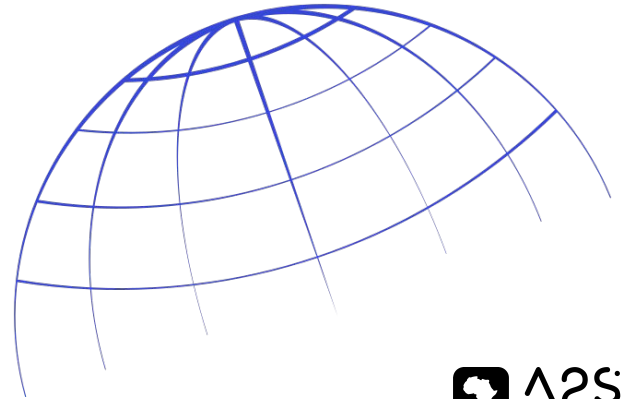
# Common Pitfalls - Mistaking Row Iterators with Column iterators

- It is pretty common to mistake your *i* and *j*'s representing rows and columns. And sometimes, especially when the matrix is a square one, it does not throw an exception but hours of confusion

```
for col_idx in range(len(matrix[0])):  
    for row_idx in range(len(matrix)):  
        print(matrix[col_idx][row_idx])
```

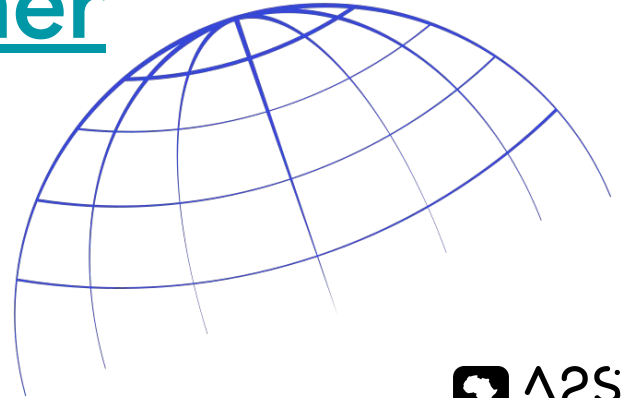


# Checkpoint



# Practice Question

## Image smoother



# Practice Questions

Reverse string

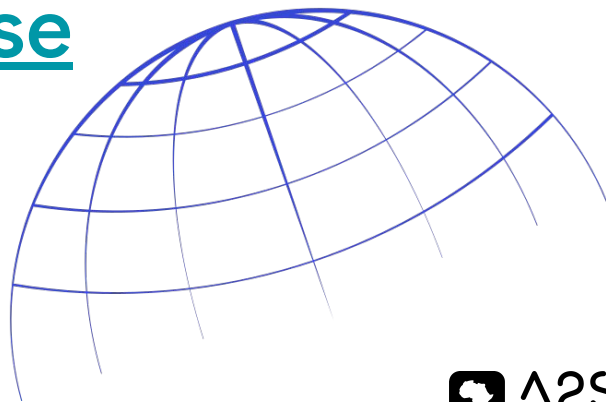
Count Equal and Divisible Pairs in an..

All Divisions With the Highest Score..

Transpose matrix

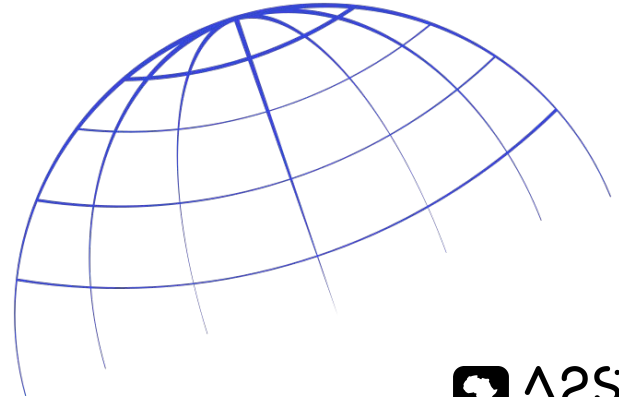
Diagonal traverse

Rotate image



# Resources

- <https://neetcode.io/courses/dsa-for-beginners/0>: useful on laying out the fundamentals
- <https://www.geeksforgeeks.org/python-arrays/>: covers large range of topics for python
- [Python List pop\(\) Method - GeeksforGeeks](#): good examples
- [Python List insert\(\) Method With Examples - GeeksforGeeks](#): good examples
- [Python | Which is faster to initialize lists? - GeeksforGeeks](#): good article on list initialization



# Quote of the Day

If you want to enjoy the rainbow,  
be prepared to endure the storm

Warren W. Wiersbe

