Advanced Programming
INFO135

Lecture 3: **Sorting algorithms** with Python (part1)

Mehdi Elahi

University of Bergen (UiB)

Call Stack

- Computers uses a stack internally called the **Call Stack**.
- Let's see it in action. Here's a simple function:

```
def greet(name):
    print('hello', name, '!')
    greet2(name)
    print('getting ready to say bye...')
    bye()
```

```
def greet2(name):
    print('how are you,' + name + '?')
```

```
def bye():
    print('ok bye!')
```

Call Stack

- This function:
  - first, **greets you**
  - and then, **calls** two **functions**: `greet2(name)` and `bye()`

```
def greet(name):
    print('hello', name, '!')
    greet2(name)
    print('getting ready to say bye...')
    bye()
```

```
greet('maggie')
```

[Output]:

```
hello maggie !
how are you,maggie?
getting ready to say bye...
ok bye!
```

Call Stack

When you call **`greet("maggie"):`**

1) first, your computer allocates **a box of memory** for that function call.

a box in memory

2) a **variable name** is set to "maggie". And saved in memory.


```
def greet(name):
    print('hello', name, '!')
    greet2(name)
    print('getting ready to say bye...')
    bye()
```

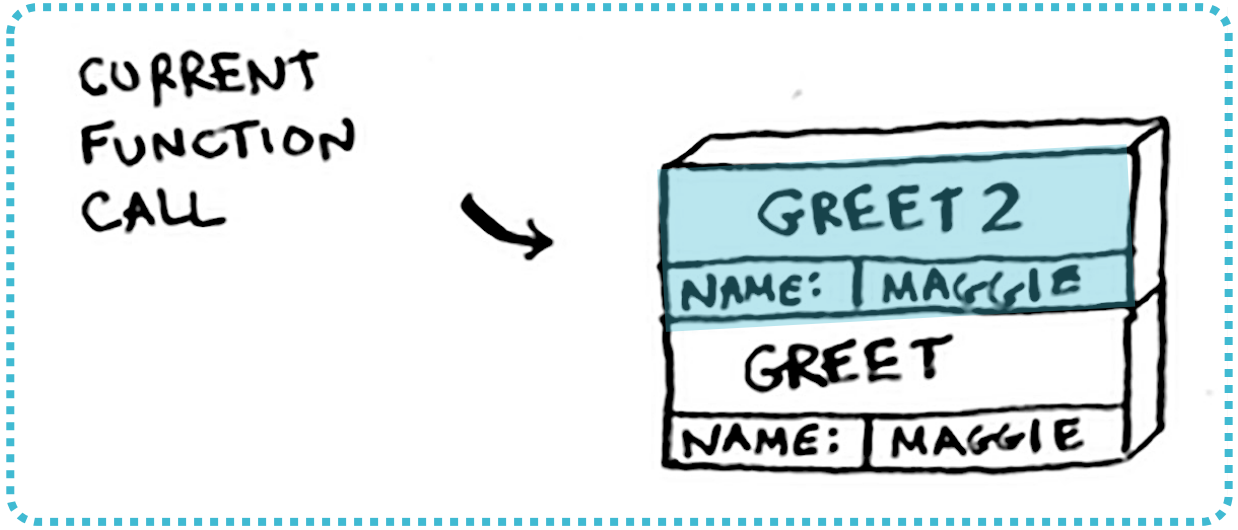
Call Stack

3) next, it prints "hello, maggie!" Then it calls **`greet2("maggie").`**

4) Again, the computer allocates the **2nd box of memory** for this function call.

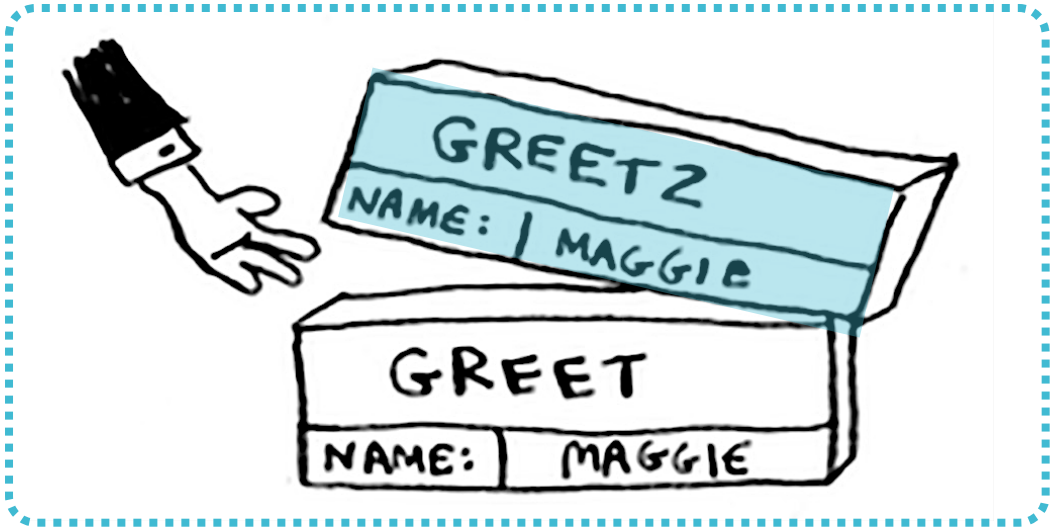
5) the 2nd box is added **on top of the 1st box**. "how are you, maggie?" is printed.

```
def greet(name):
    print('hello', name, '!')
    greet2(name)
    print('getting ready to say bye...')
    bye()
```



Call Stack

5) then, it returns from the function call, and the box on top gets **popped** off.




```
def greet(name):
    print('hello', name, '!')
    greet2(name)
    print('getting ready to say bye...')
    bye()
```

Call Stack

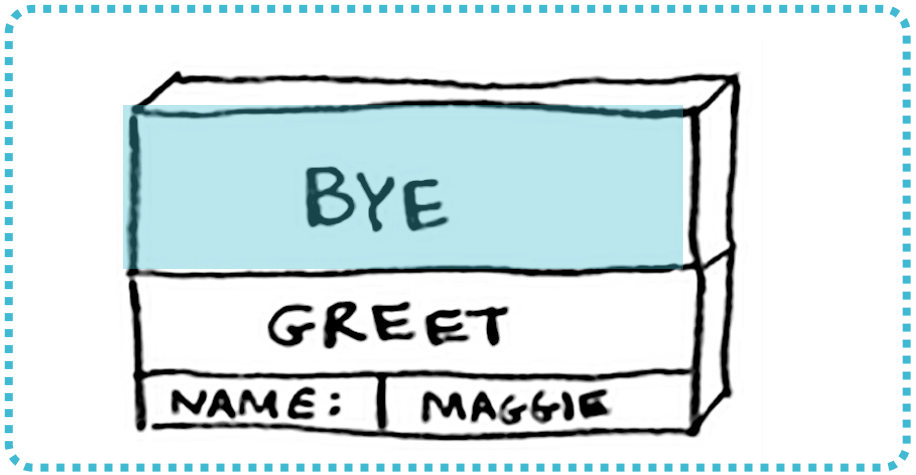
7) Now that it is done with the **`greet2()`** function, and back to the **`greet()`** function. Then it picks up where it left off.

8) Now it prints "getting ready to say bye…." and

```
def greet(name):
    print('hello', name, '!')
    greet2(name)
    print('getting ready to say bye...')
    bye()
```

Call Stack

8) And **`bye()`** function is called. The 3rd box for this function is added to the top of the stack.




```
def greet(name):
    print('hello', name, '!')
    greet2(name)
    print('getting ready to say bye...')
    bye()
```

Call Stack

9) finally, it prints "ok bye!" and return from the **function call**.


```
def greet(name):
    print('hello', name, '!')
    greet2(name)
    print('getting ready to say bye...')
    bye()
```

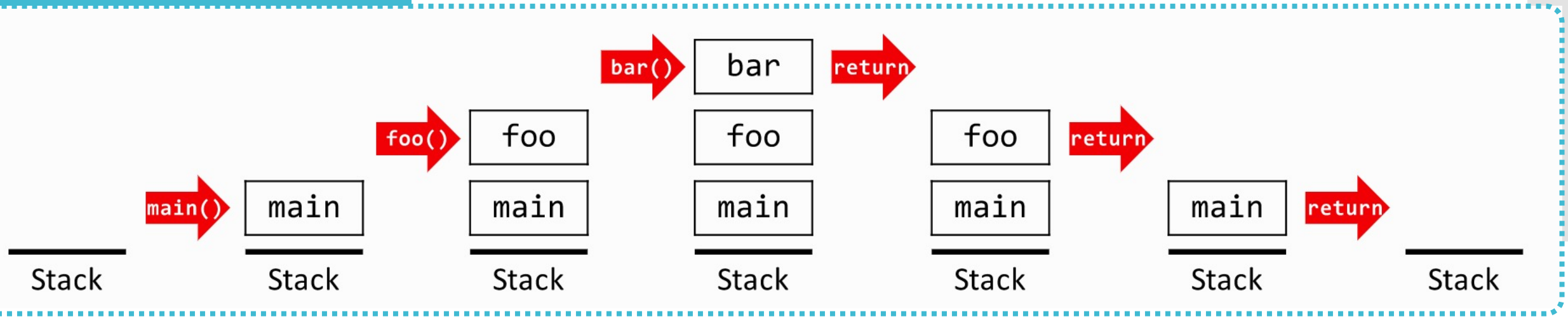
Quiz

- Draw the Call Stack for the following function calls:

```
def main_f():
    foo()

def foo():
    bar()

def bar():
    pass
```

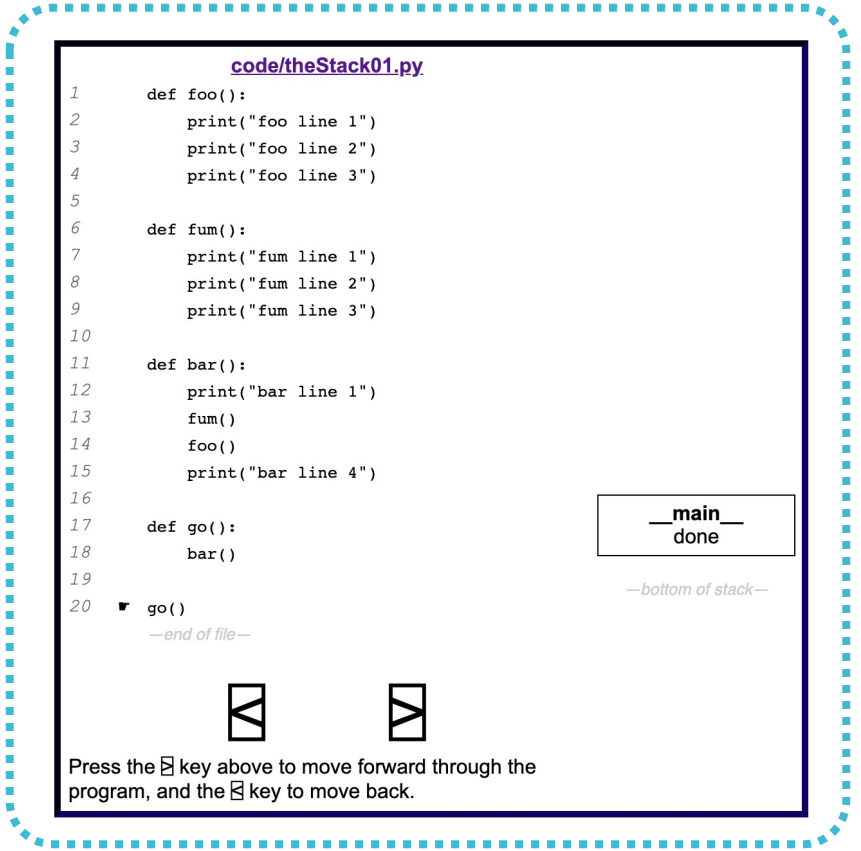
Answer

```
def main_f():
    foo()

def foo():
    bar()

def bar():
    pass
```



Call Stack

- Another interactive example:
  - **Link:** https://sites.cs.ucsb.edu/~pconrad/cs8/topics.beta/theStack/02/
  - **Short link:** http://bit.do/call_stack

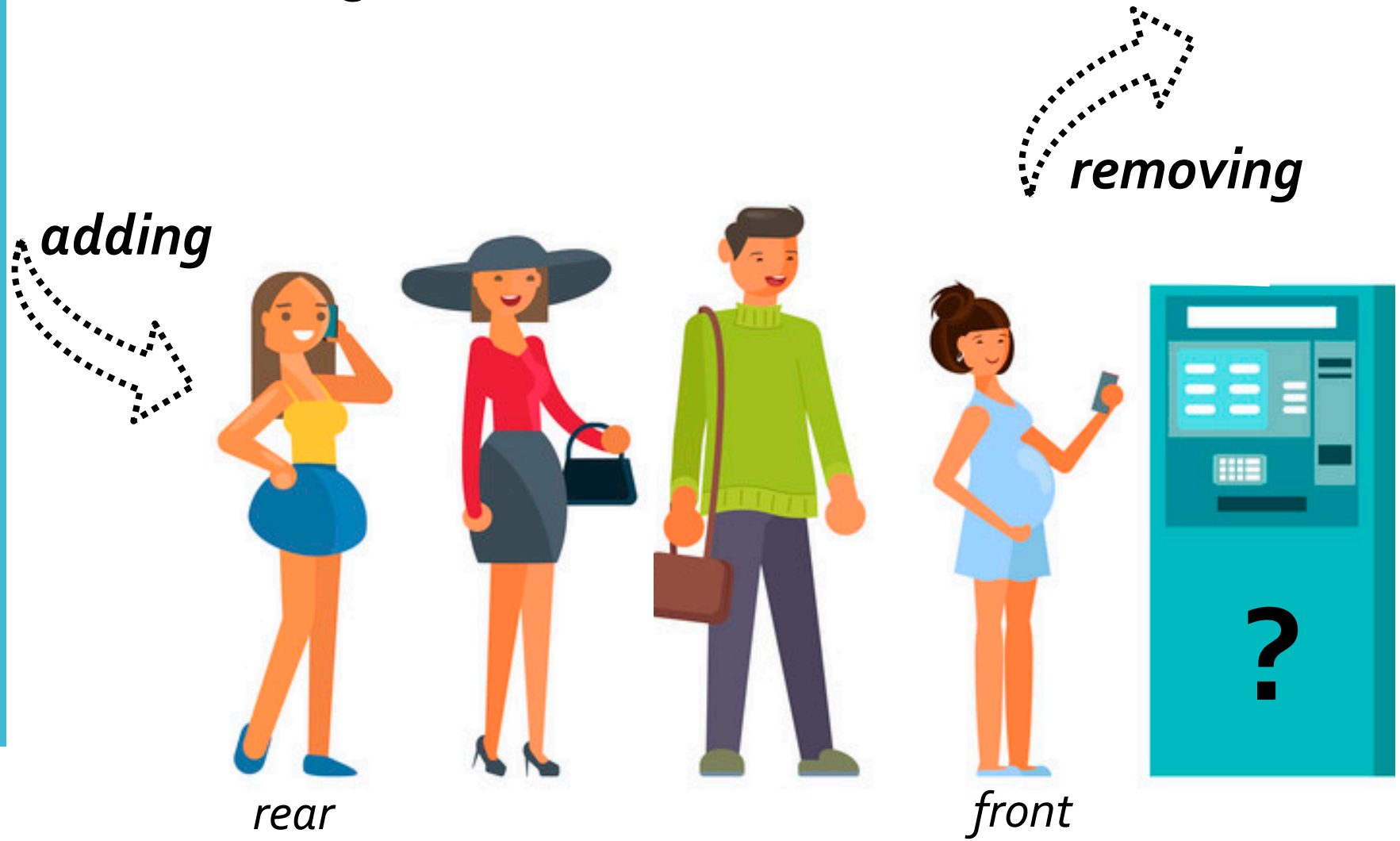


Queue

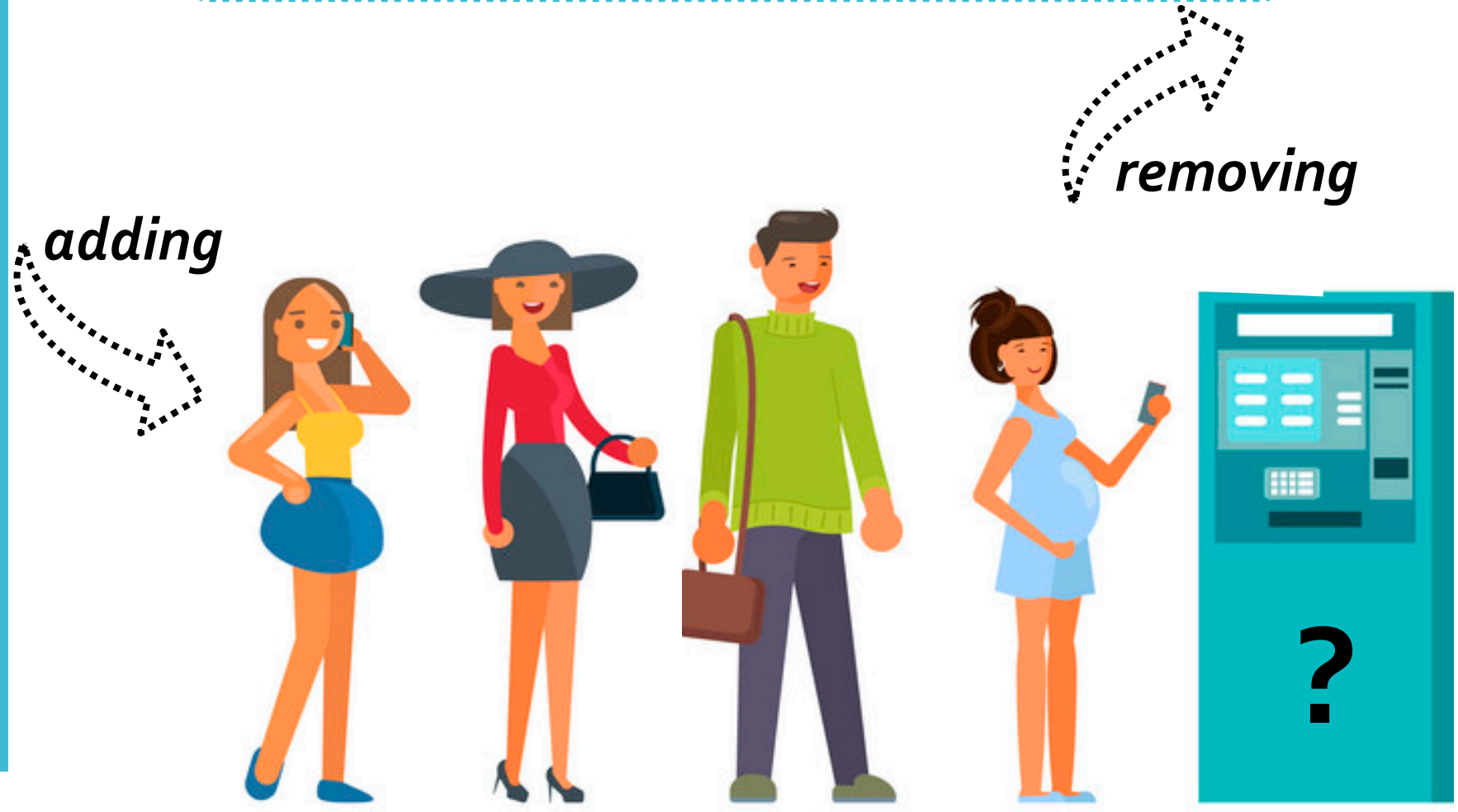
- **Queue** of people.
  - how a new person is **added**?
  - how a person is **removed**?

Queue

- **Queue:** an ordered collection of elements where:
  - new elements are **added** at the **rear**
  - existing elements are removed from **front**.



Queue

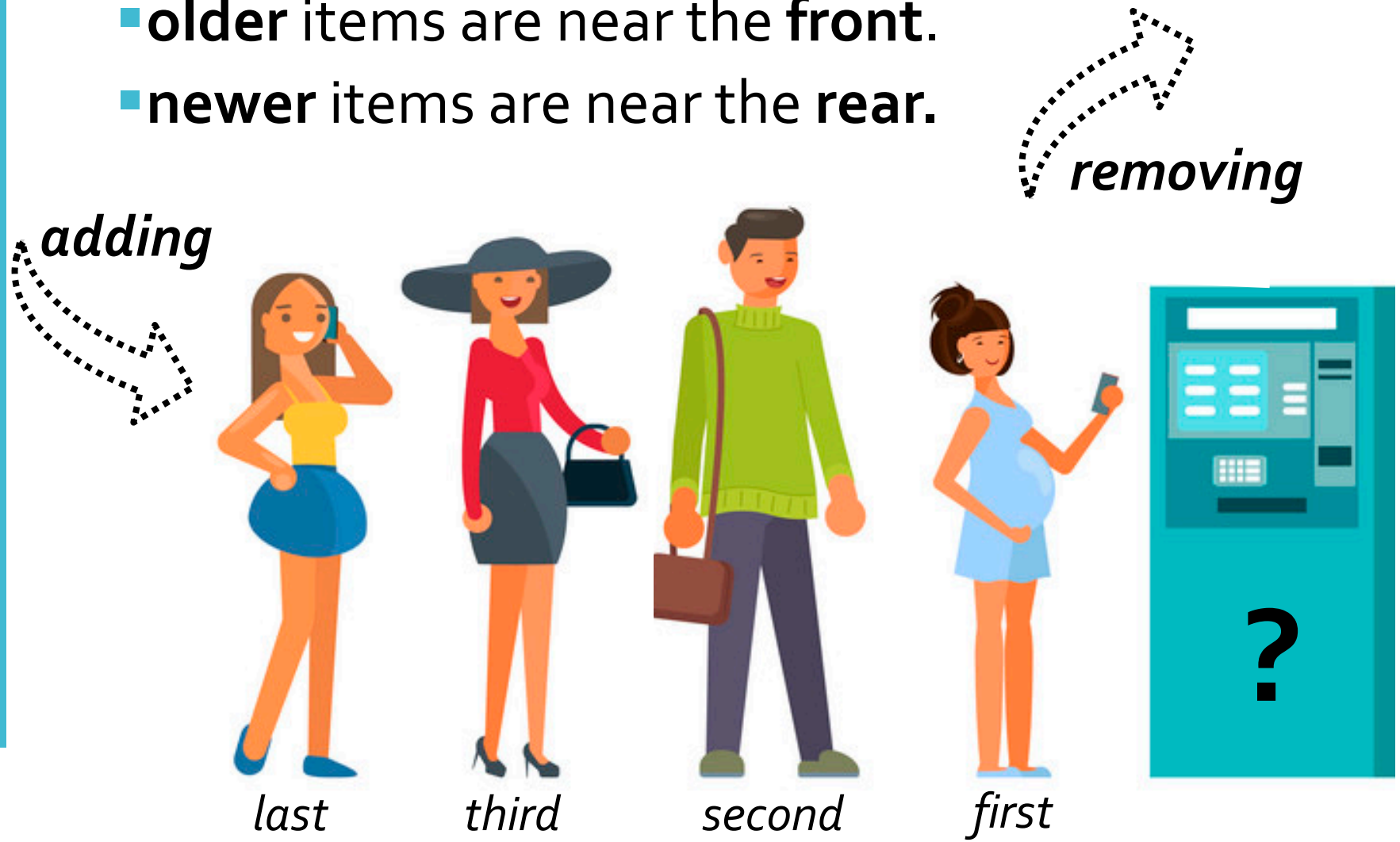


Queue

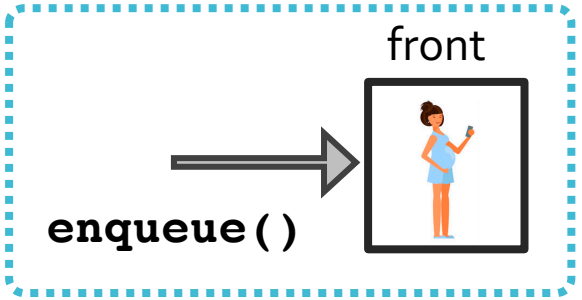
- Most **recent element** waits **at the end** of the collection.
- Element that waits in the collection **the longest** is **at the front**.

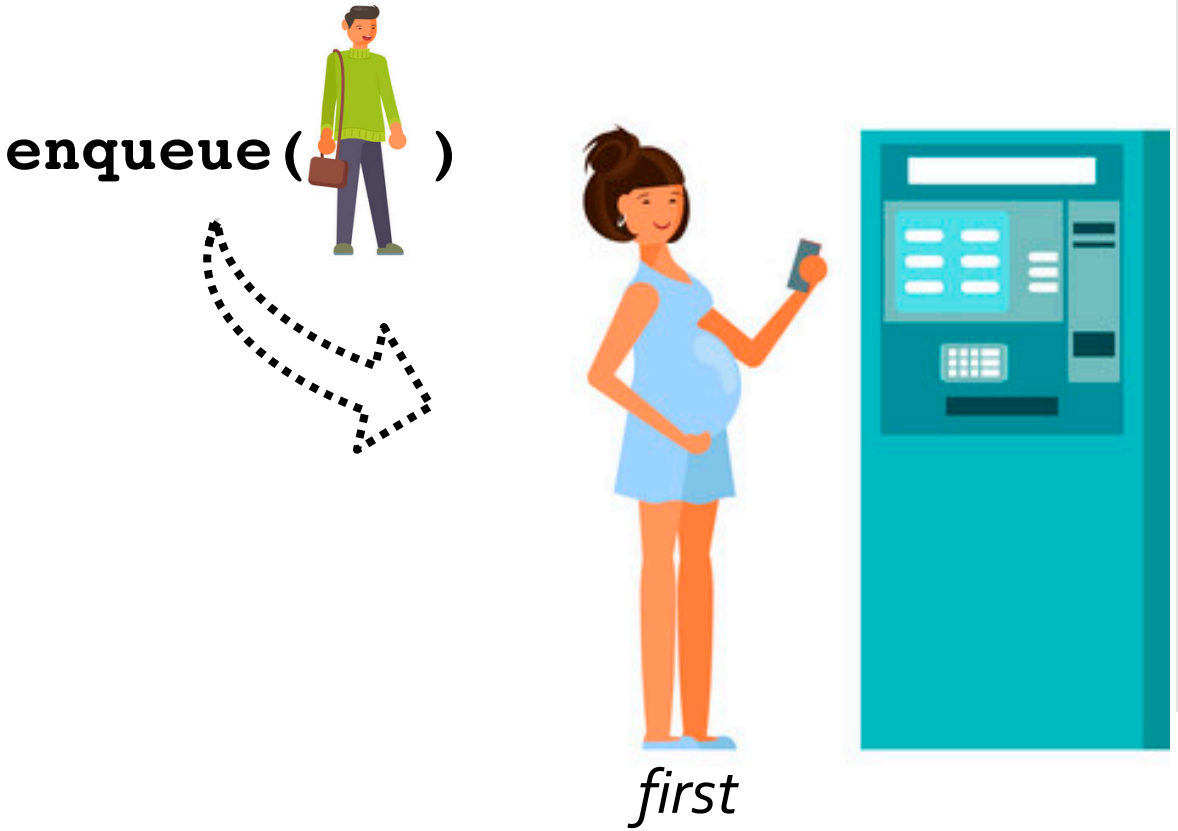
Queue

- **First In First Out (FIFO)**:
  - **least recently** added item is **removed first**.
  - **older** items are near the **front**.
  - **newer** items are near the **rear.**

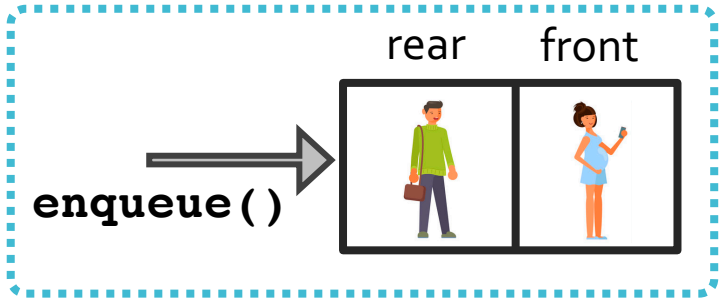


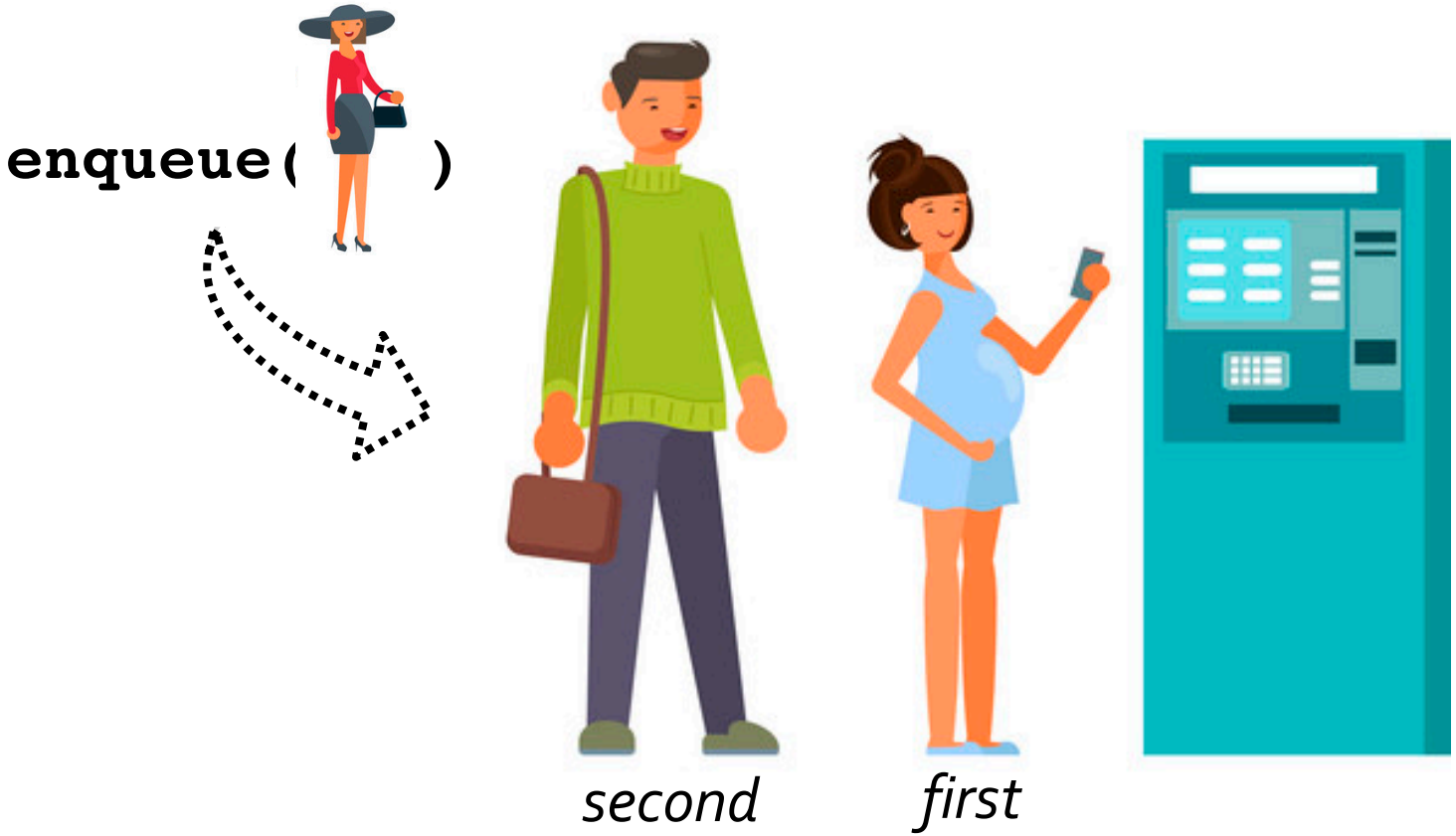
Queue



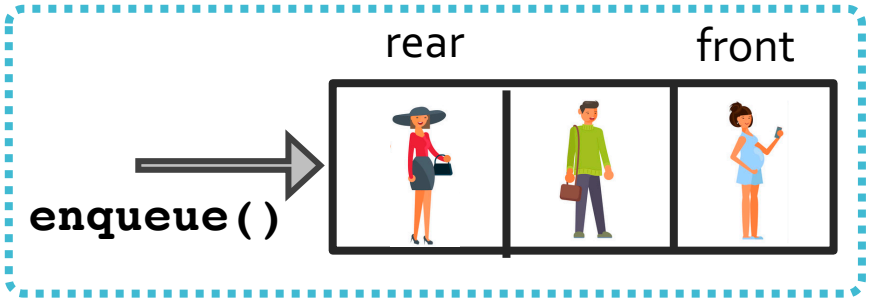


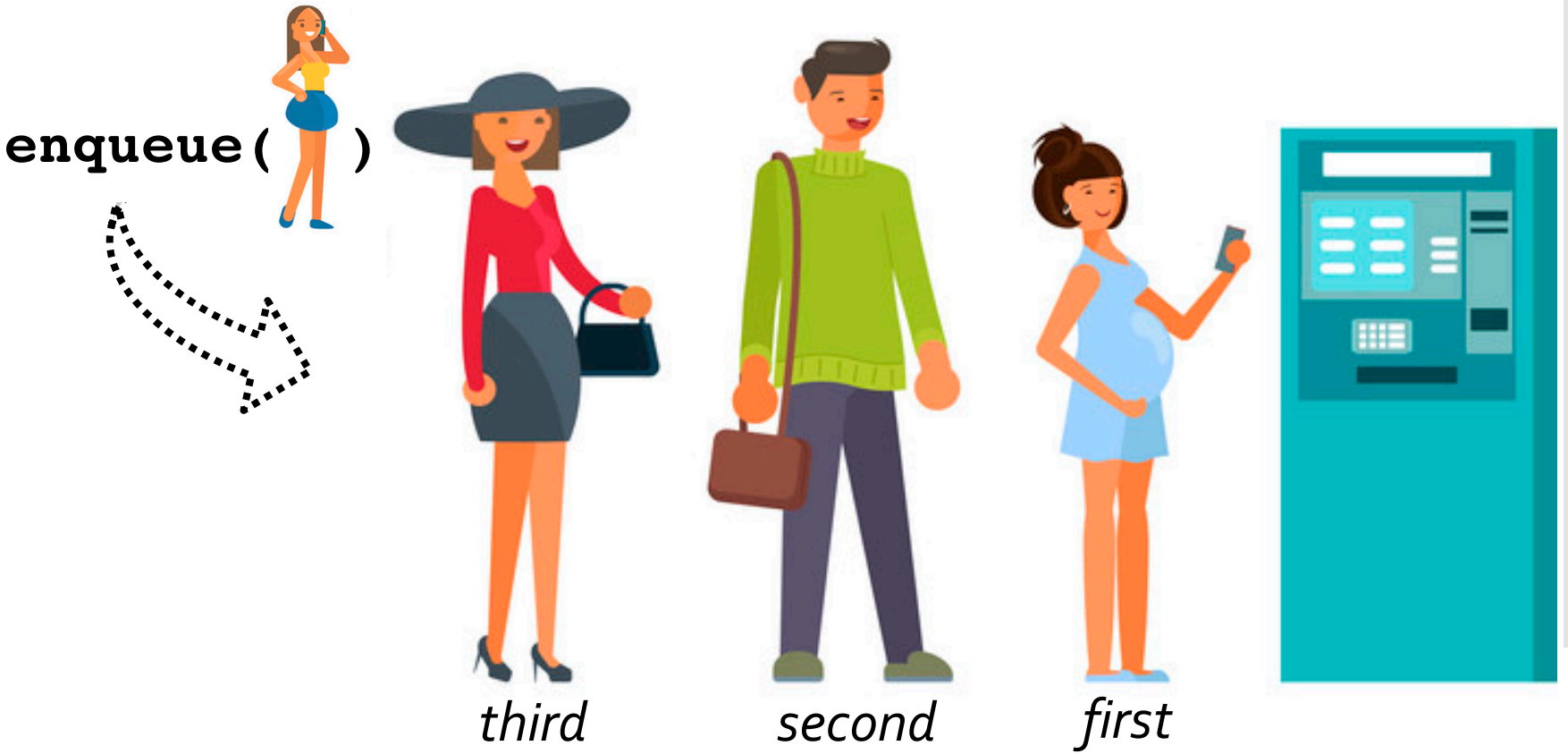
Queue



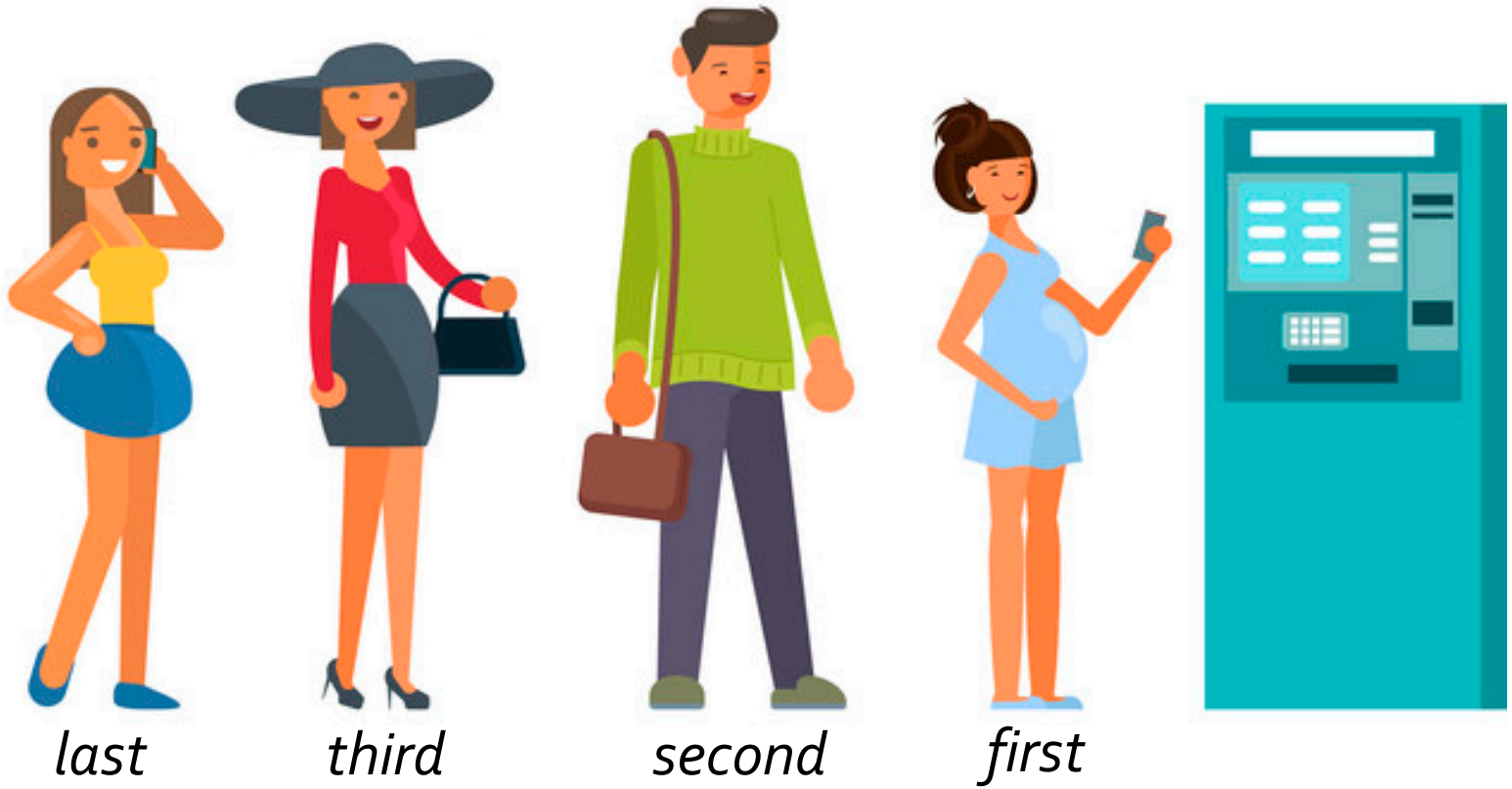


Queue

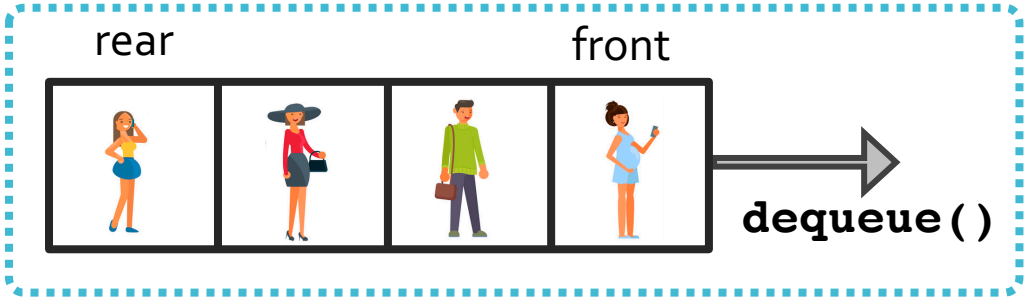


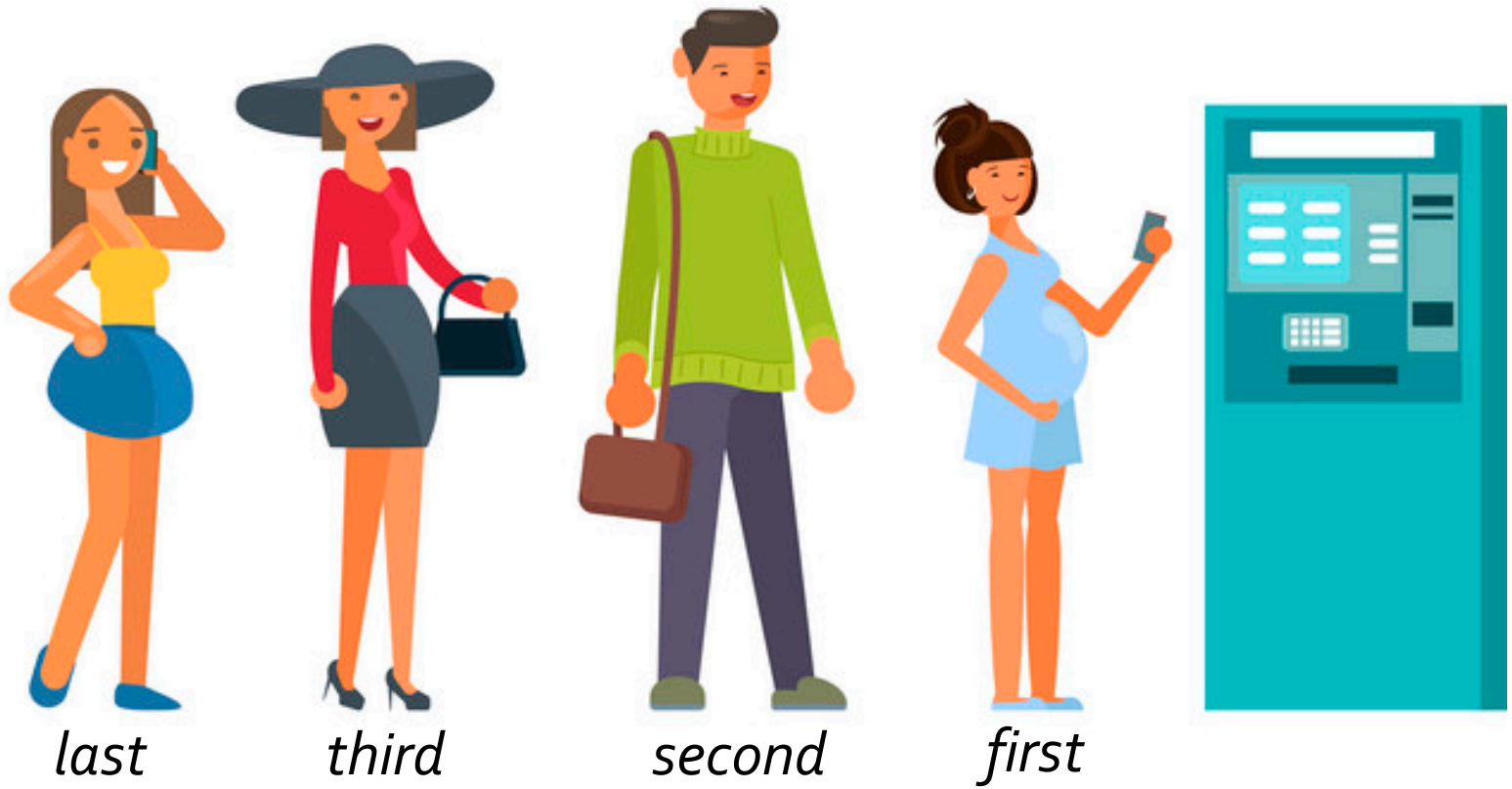


Queue

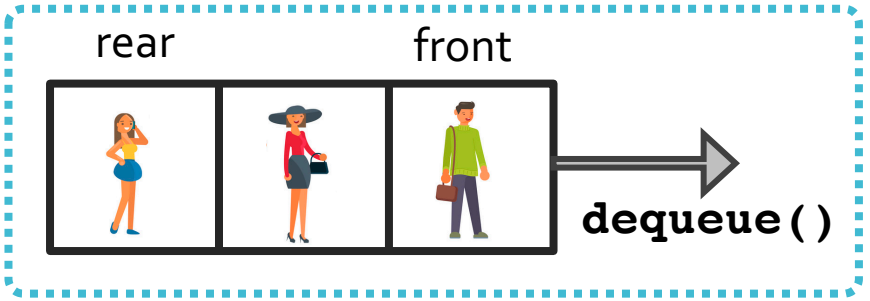


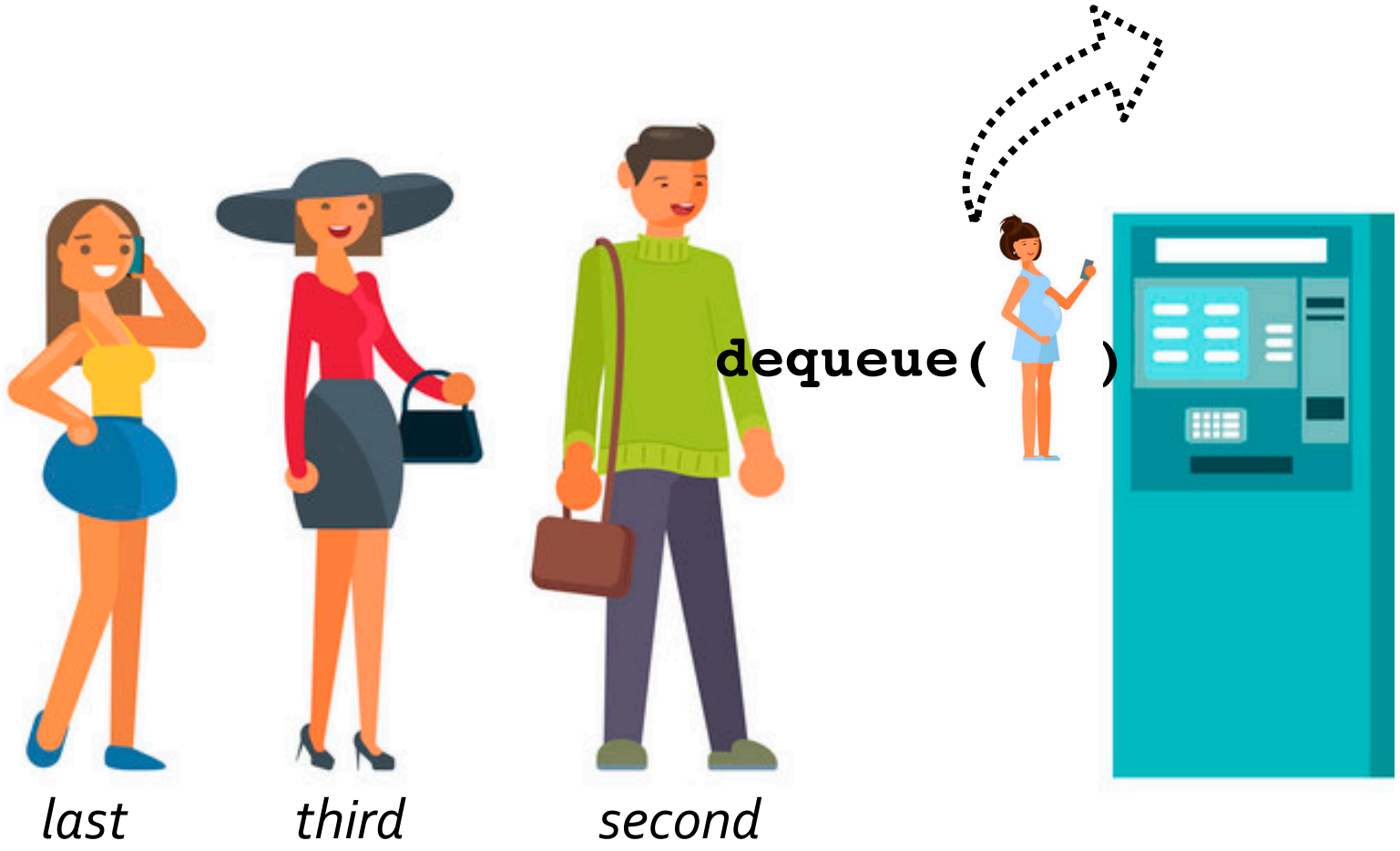
Queue



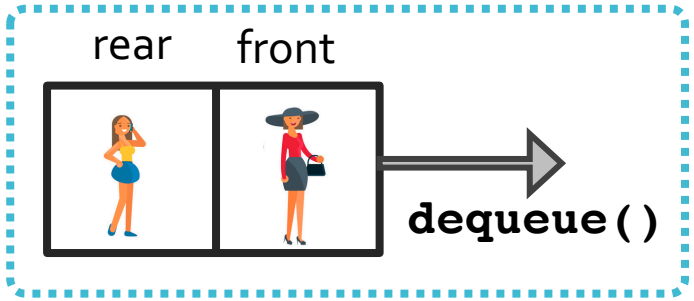


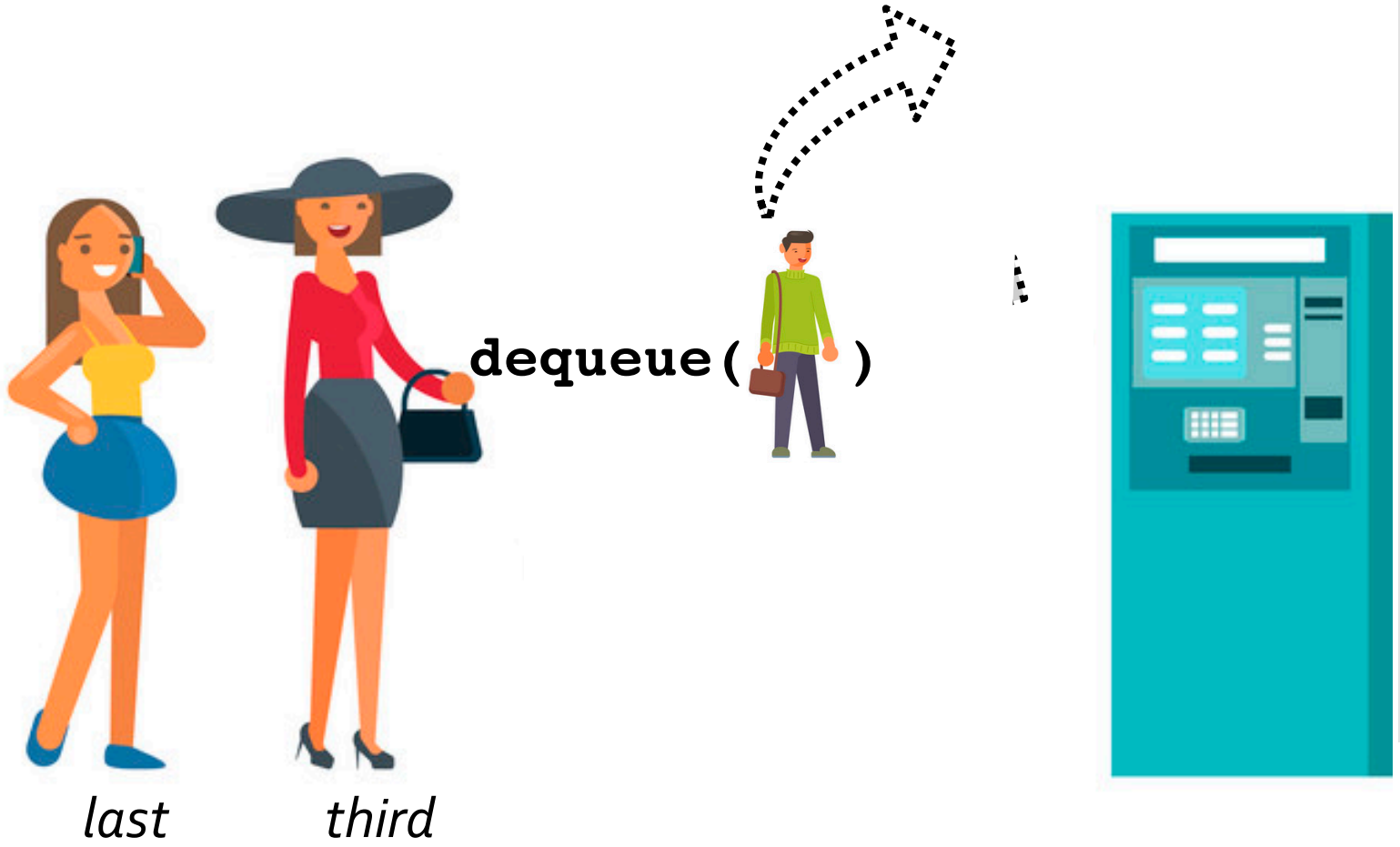
Queue



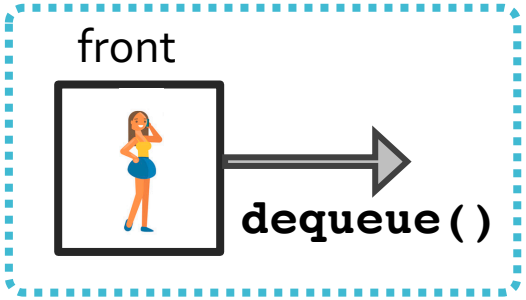


Queue





Queue



Queue ADT

- **`Queue()`:**
  - Creates a new empty queue.
- **`is_empty()`:**
  - Returns a boolean value indicating whether or not the queue is empty.
- **`size()`:**
  - Returns the number of items in the queue.
- **`enqueue(item)`:**
  - Adds the given item to the back of the queue.
- **`dequeue()`:**
  - Removes and returns the front item of the queue.

Queue

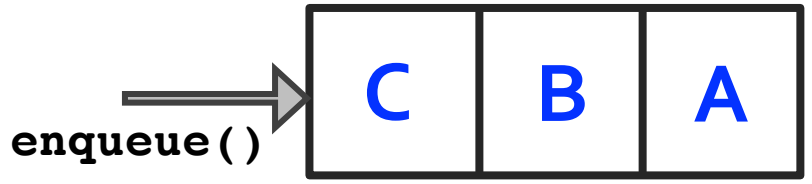
- Implementing **Queue** ADT with **Python list**

```
class Queue:
    def __init__(self):
        self.items = []

    def is_empty(self):
        return self.items == []

    def enqueue(self, item):
        self.items.insert(0, item)

    def dequeue(self):
        return self.items.pop()

    def size(self):
        return len(self.items)
```

Queue

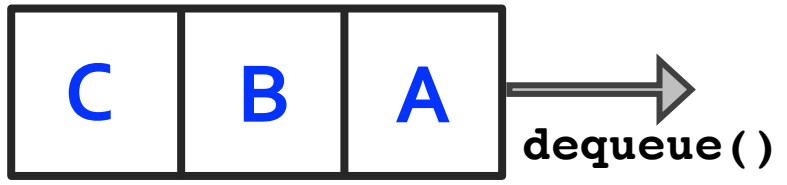
❖ Building a **Queue** and calling the methods.

```
my_queue = Queue()
my_queue.enqueue('A')
my_queue.enqueue('B')
my_queue.enqueue('C')

print("size of the queue:", my_queue.size())
print()
print("first  element:", my_queue.dequeue())
print("second element:", my_queue.dequeue())
print("third  element:", my_queue.dequeue())
```



[Output]:

```
size of the queue: 3

first  element: A
second element: B
third  element: C
```



Doubly Linked-list

- We can also implement **Queue** ADT with **Doubly** Linked-list

Singly Linked-list

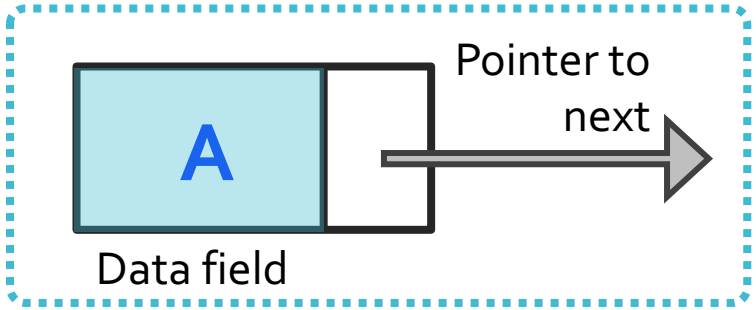
- Remember the implementation of a **Node**

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Data field

Pointer

Node



Doubly Linked-list

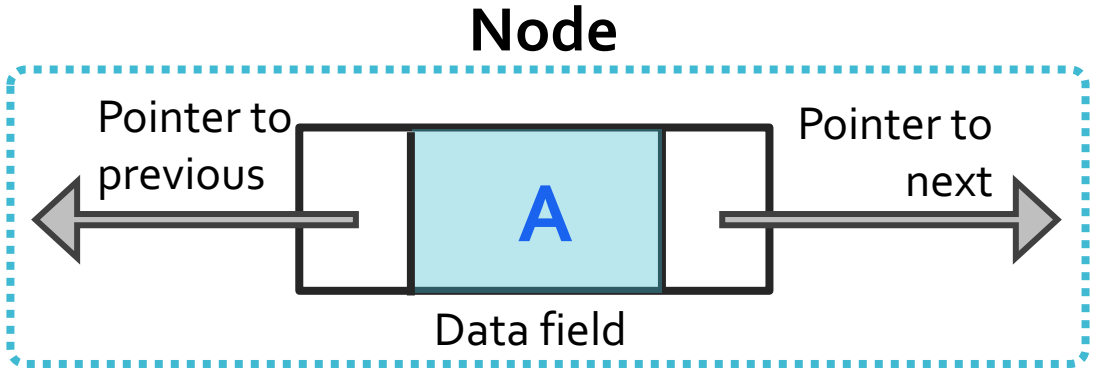
- Here is an extended version of **Node**

```
class Node:

    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None
```

Data field

Pointer to **next**

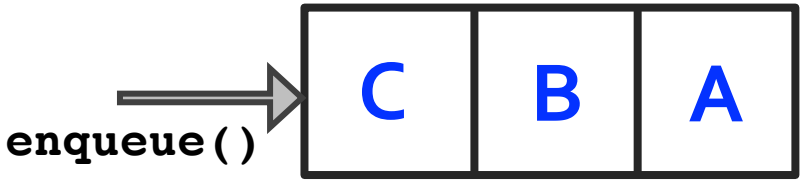
Pointer to **previou**



Queue

❖ Implementing **Queue** ADT with **doubly linked-list**

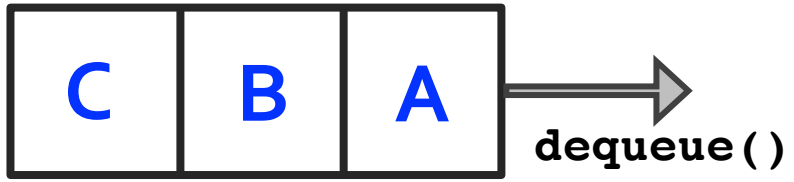
```
class Queue:
    def __init__(self):
        self.head = None
        self.last = None
    def enqueue(self, data):
        if self.last is None:
            self.head = Node(data)
            self.last = self.head
        else:
            self.last.next = Node(data)
            self.last.next.prev = self.last
            self.last = self.last.next
```



Queue

- Implementing **Queue** ADT with **doubly linked list**

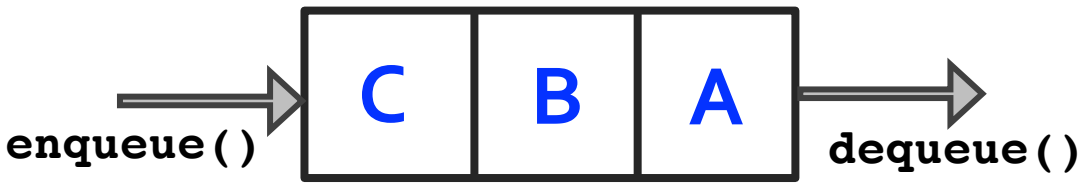
```
class Queue:
    def __init__(self):
        self.head = None
        self.last = None
    def dequeue(self):
        if self.head is None:
            return None
        else:
            temp = self.head.data
            self.head = self.head.next
            self.head.prev = None
            return temp
```



Queue

❖ Implementing **Queue** ADT with **doubly linked list**

```
class Queue:
    def __init__(self):
        self.head = None
        self.last = None

    def size(self):
        temp = self.head
        count = 0
        while temp is not None:
            count = count + 1
            temp = temp.next
        return count
```



Quiz

- Write a function **`reverse_queue()`** that receives a queue, builds a **stack** with the same elements, and returns a reversed queue.

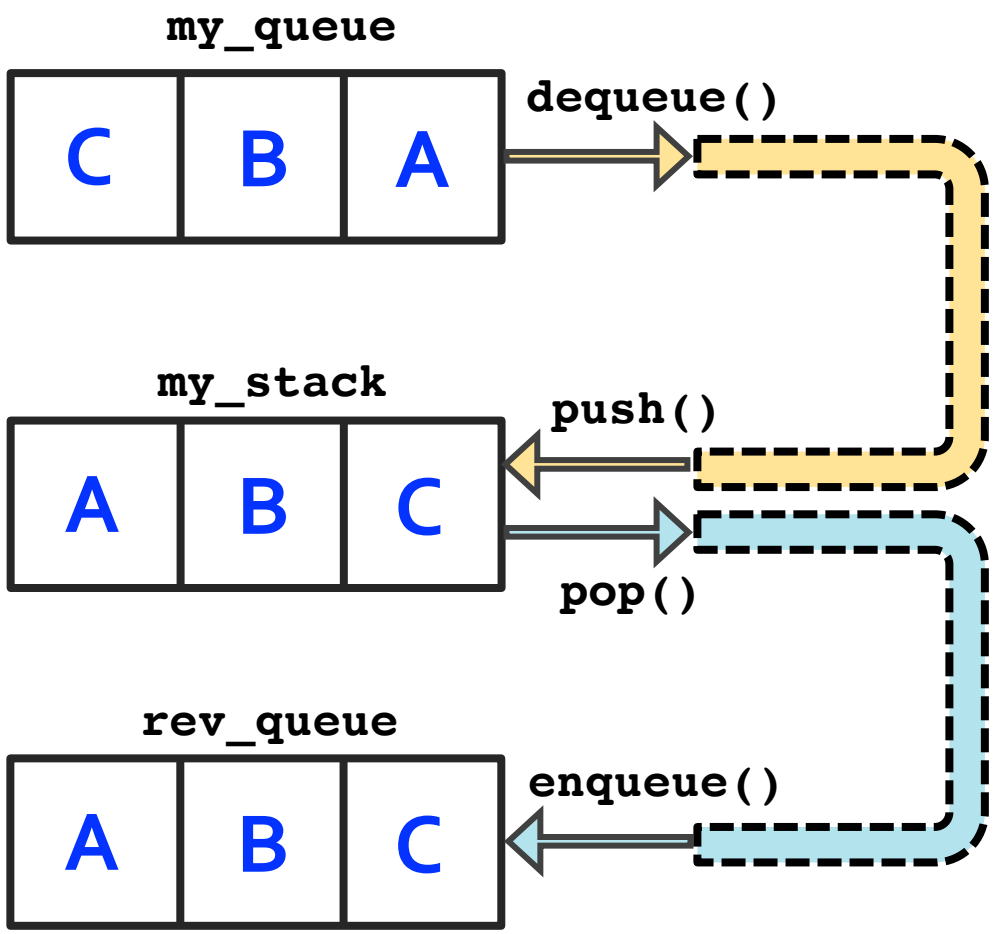
```
my_queue = Queue()
my_queue.enqueue('A')
my_queue.enqueue('B')
my_queue.enqueue('C')

rev_queue = reverse_queue(my_queue)

print("first  element:", rev_queue.dequeue())
print("second element:", rev_queue.dequeue())
print("third  element:", rev_queue.dequeue())
```

[Output]:

```
first  element: C
second element: B
third  element: A
```

Quiz

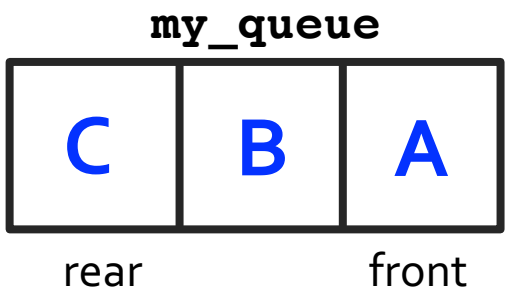
- Write a function **`reverse_queue()`** that receives a queue, builds a **stack** with the same elements, and returns a reversed queue.



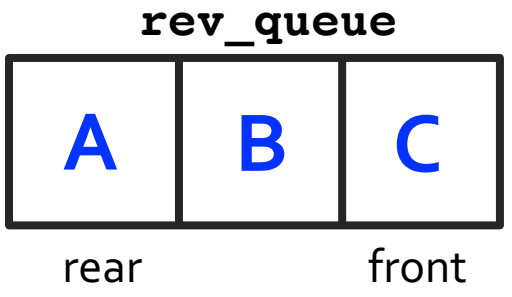
Answer

```
def reverse_queue(queue):
    my_stack = Stack()
    rev_queue = Queue()

    while not queue.is_empty():
        my_stack.push(queue.dequeue())

    while not my_stack.is_empty():
        rev_queue.enqueue(my_stack.pop())

    return rev_queue
```

[Input]:



[Output]:



Sorting

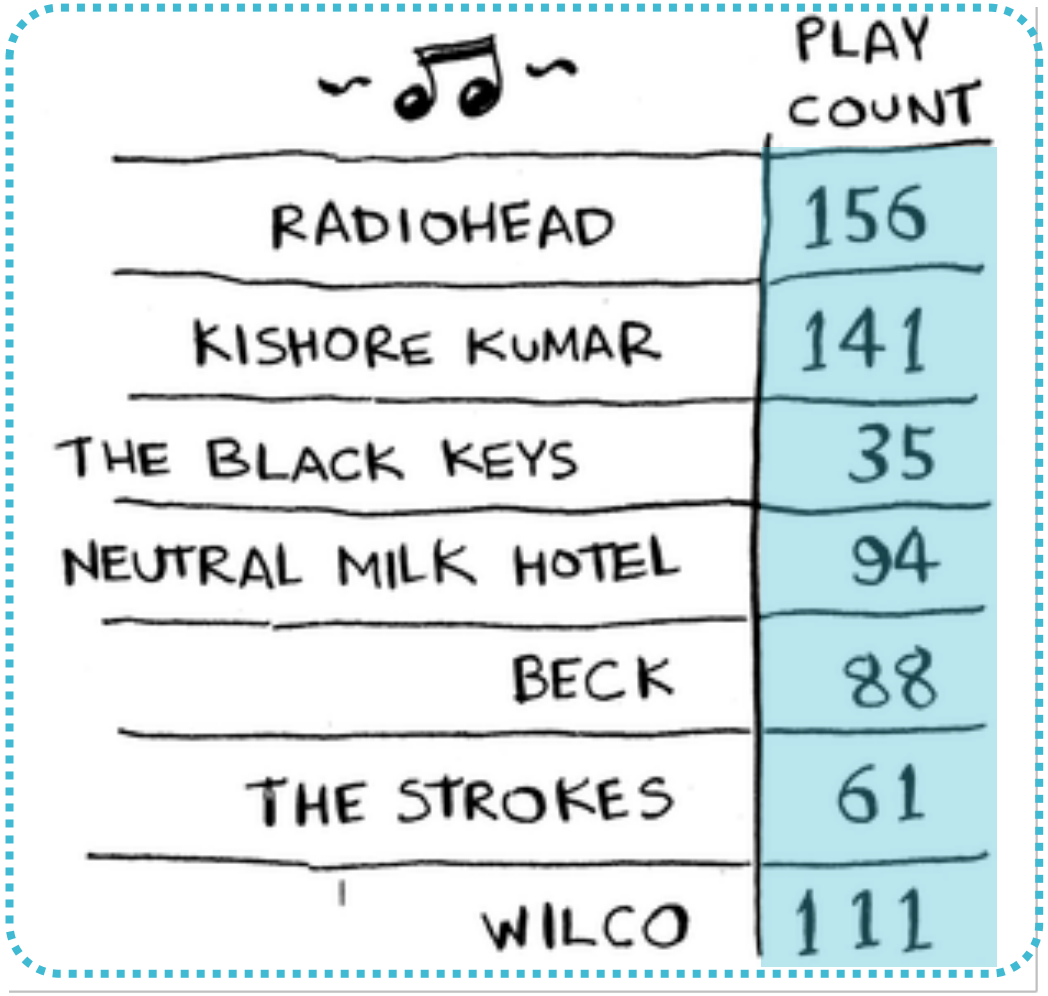
- **Sorting** is the process of placing elements from a collection in some kind of order.

- **Examples:**
  - a list of **music** songs could be **sorted** according to the number of times they have been played (**#count**).
  - a list of **cities** could be **sorted** by **population**, or by zip code.

- **Efficiency** of a sorting algorithm is related to the number of items being processed.

Selection Sort

- Suppose you have a bunch of music and you know the **play count** for each artist.
- How to **sort** the list from **most to least** played**?**

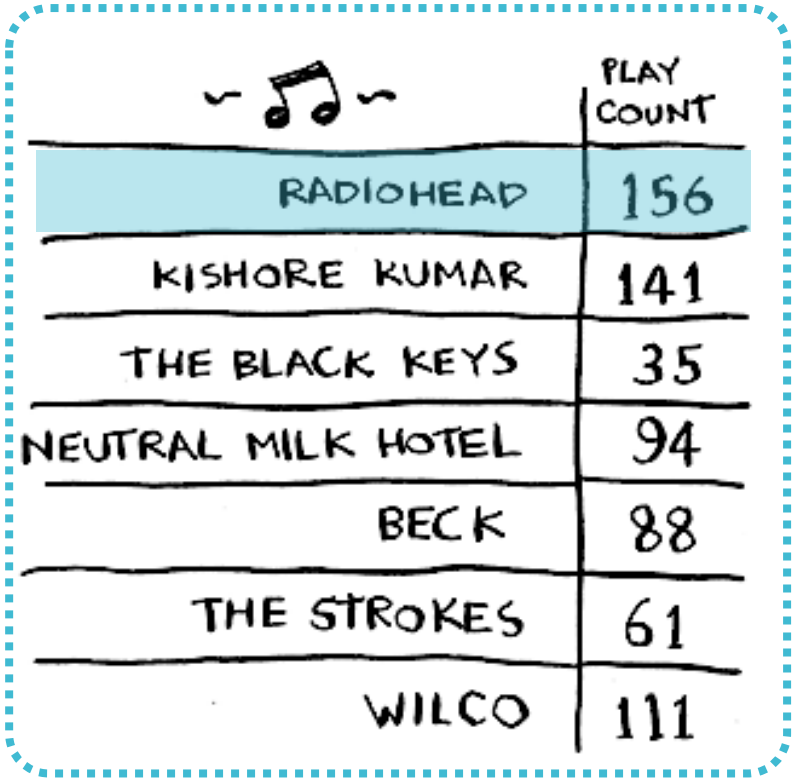


	PLAY COUNT
RADIOHEAD	156
KISHORE KUMAR	141
THE BLACK KEYS	35
NEUTRAL MILK HOTEL	94
BECK	88
THE STROKES	61
WILCO	111

Not sorted

Selection Sort

- You can go through the list and find **most-played** artist.
- **Add** that artist to a **new list**.



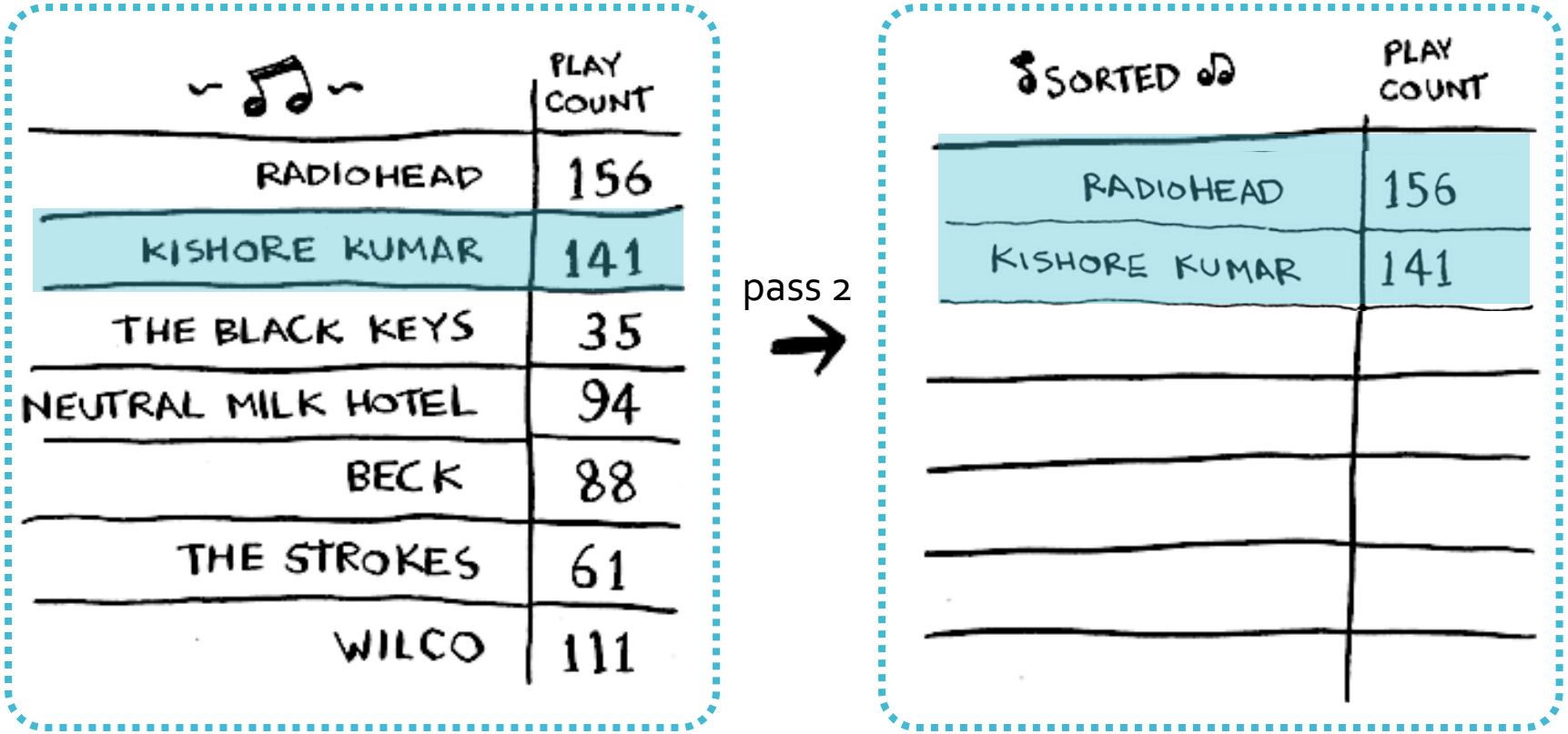
pass 1

1. RADIOHEAD IS THE MOST PLAYED ARTIST...

2. ADD IT TO A NEW LIST

Selection Sort

- You can **repeat** to find next-most-played artist.
- **Add** that artist to a **new list**.



Selection Sort

- You can **repeat** to find next-most-played artist.
- **Add** that artist to a **new list**.

♪	PLAY COUNT
RADIOHEAD	156
KISHORE KUMAR	141
THE BLACK KEYS	35
NEUTRAL MILK HOTEL	94
BECK	88
THE STROKES	61
WILCO	111

pass 3 →

♪ SORTED ♪	PLAY COUNT
RADIOHEAD	156
KISHORE KUMAR	141
WILCO	111

Selection Sort

- You can **repeat** to find next-most-played artist.
- **Add** that artist to a **new list**.

	PLAY COUNT
RADIOHEAD	156
KISHORE KUMAR	141
THE BLACK KEYS	35
NEUTRAL MILK HOTEL	94
BECK	88
THE STROKES	61
WILCO	111

pass 4 →

SORTED	PLAY COUNT
RADIOHEAD	156
KISHORE KUMAR	141
WILCO	111
NEUTRAL MILK HOTEL	94

Selection Sort

- You can **repeat** to find next-most-played artist.
- **Add** that artist to a **new list**.

	PLAY COUNT
RADIOHEAD	156
KISHORE KUMAR	141
THE BLACK KEYS	35
NEUTRAL MILK HOTEL	94
BECK	88
THE STROKES	61
WILCO	111

pass 5 →

SORTED	PLAY COUNT
RADIOHEAD	156
KISHORE KUMAR	141
WILCO	111
NEUTRAL MILK HOTEL	94
BECK	88

Selection Sort

- You can **repeat** to find next-most-played artist.
- **Add** that artist to a **new list**.

	PLAY COUNT
RADIOHEAD	156
KISHORE KUMAR	141
THE BLACK KEYS	35
NEUTRAL MILK HOTEL	94
BECK	88
THE STROKES	61
WILCO	111

pass 6 →

SORTED	PLAY COUNT
RADIOHEAD	156
KISHORE KUMAR	141
WILCO	111
NEUTRAL MILK HOTEL	94
BECK	88
THE STROKES	61

Selection Sort

- You can **repeat** to find next-most-played artist.
- **Add** that artist to a **new list**.

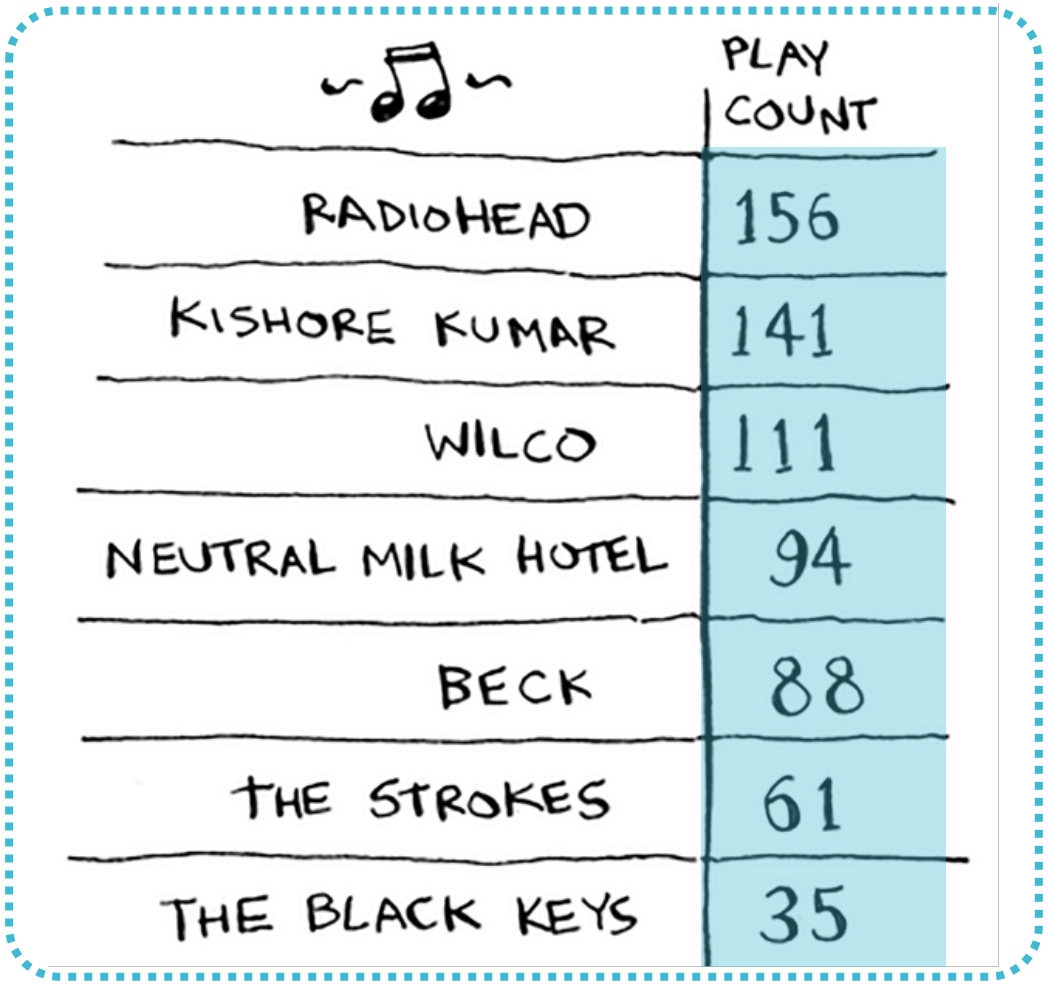
	PLAY COUNT
RADIOHEAD	156
KISHORE KUMAR	141
THE BLACK KEYS	35
NEUTRAL MILK HOTEL	94
BECK	88
THE STROKES	61
WILCO	111

pass 7 →

SORTED	PLAY COUNT
RADIOHEAD	156
KISHORE KUMAR	141
WILCO	111
NEUTRAL MILK HOTEL	94
BECK	88
THE STROKES	61
THE BLACK KEYS	35

Selection Sort

- Finally, you end up with a **sorted list**.



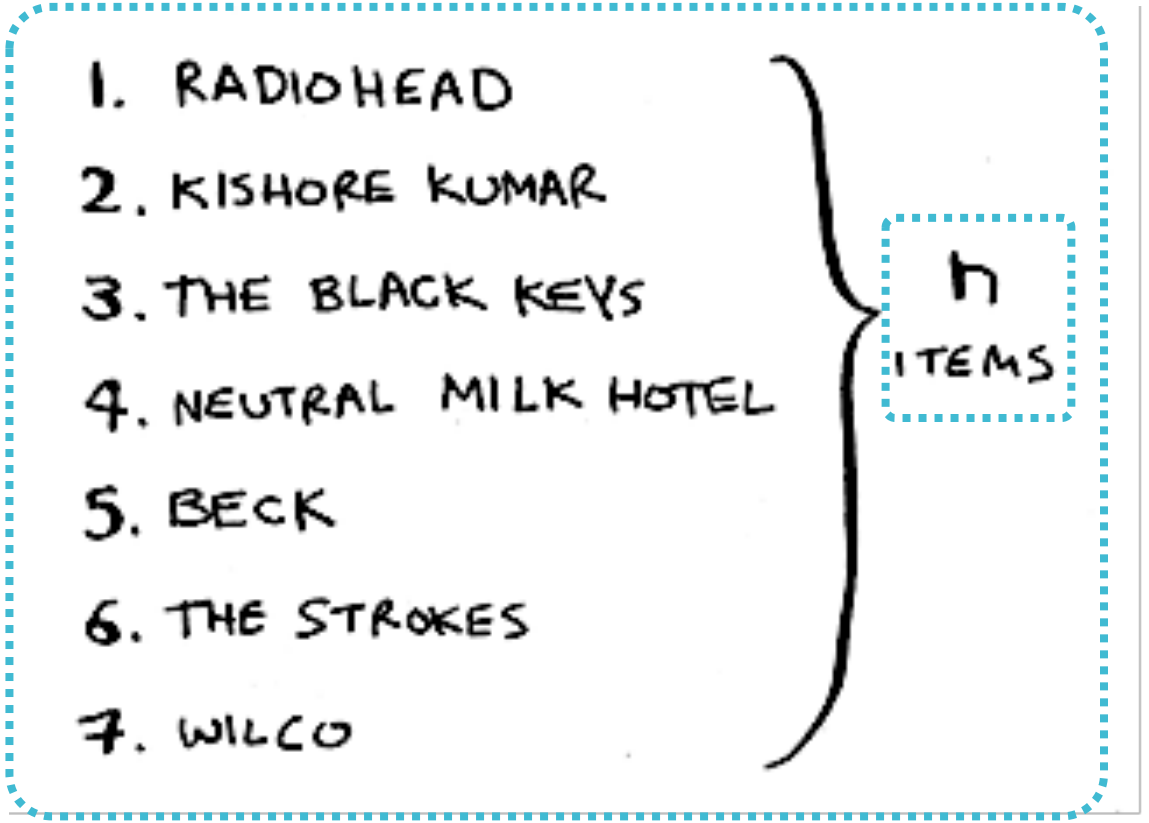
	PLAY COUNT
RADIOHEAD	156
KISHORE KUMAR	141
WILCO	111
NEUTRAL MILK HOTEL	94
BECK	88
THE STROKES	61
THE BLACK KEYS	35

7 items
7 passes

for **n items?**

Selection Sort

- **How long** this will take to run for **n items**?
- As you know, **O(n)** time means to touch **every element** in a list **once**.

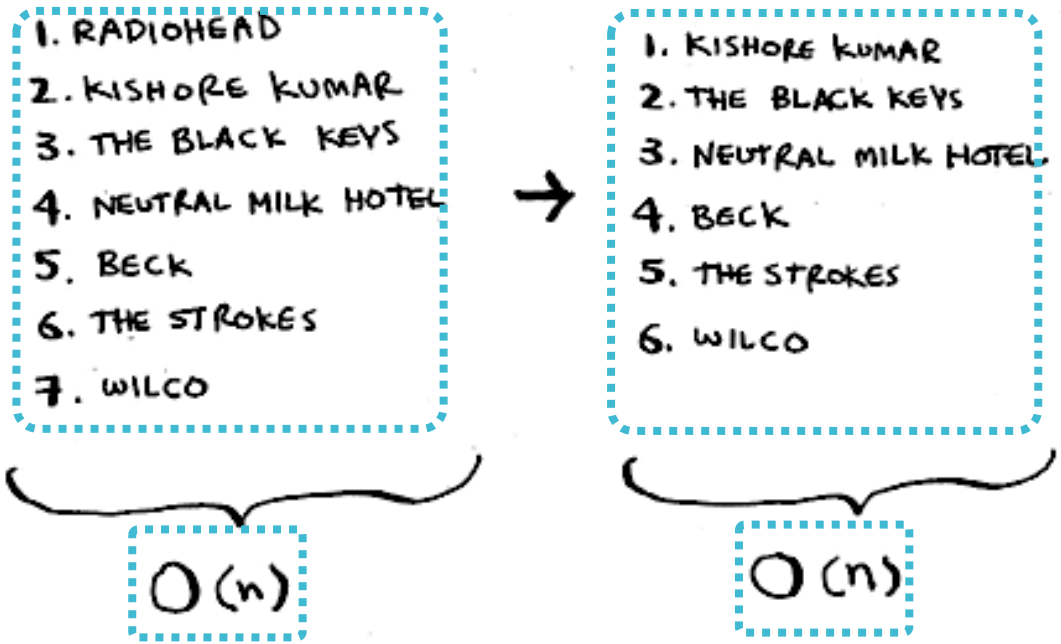


Selection Sort

- To find the artist with the highest play count:
  - you have to **check each item** in the list.
- This operation takes **O(n) time**

Selection Sort

- To find the artist with the highest play count:
  - you have to **check each item** in the list.
- This operation also takes **O(n) time**

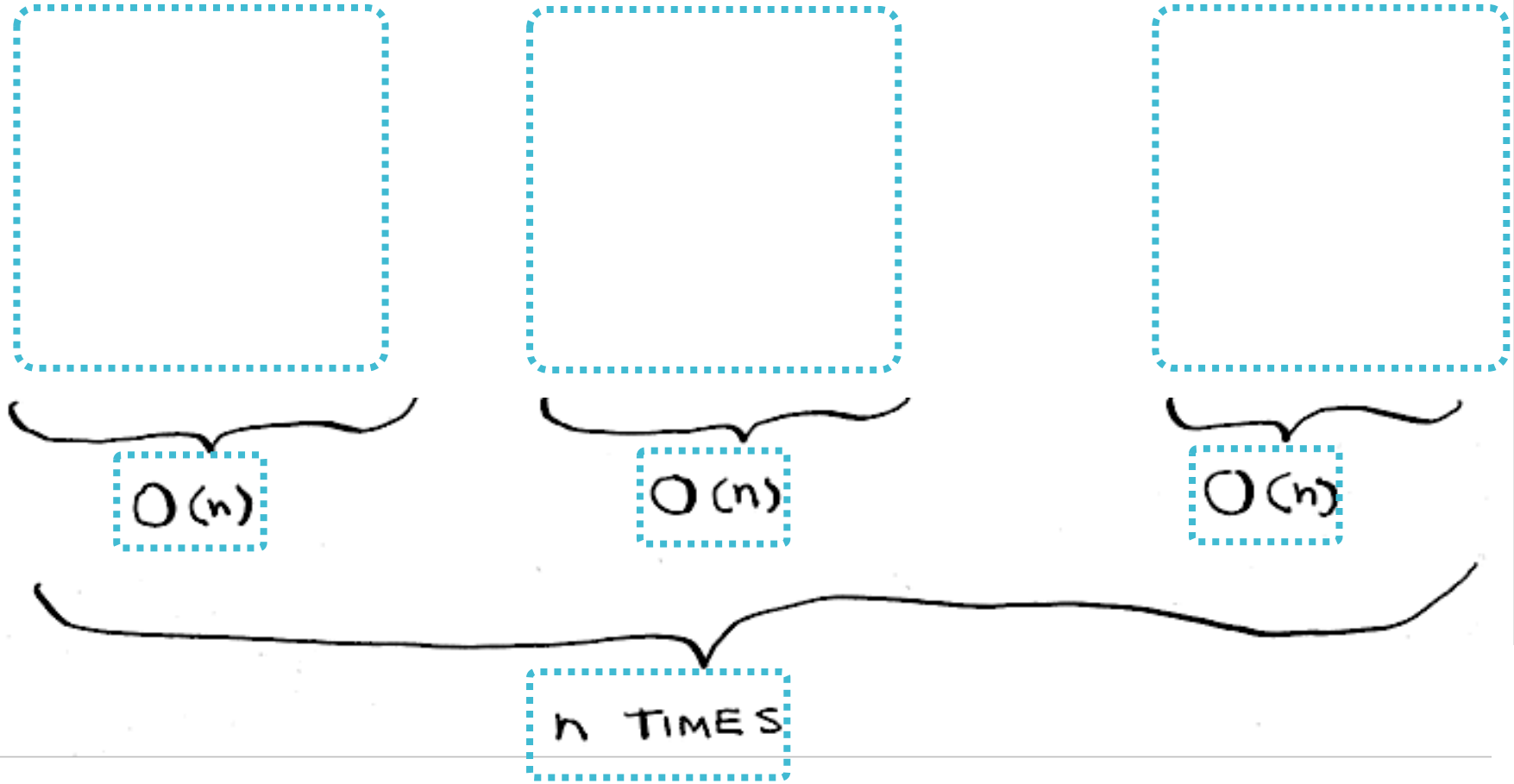


Selection Sort

- To find the artist with the highest play count:
  - you have to **check each item** in the list.
- And this operation again takes **O(n) time**

Selection Sort

- Selection Sort algorithm:
- takes O(n × n) time = **O(n²) time**.



Selection Sort

- Selection Sort algorithm:
- takes O(n × n) time = **O(n²) time**.

	Best case	Average case	Worse case
Selection Sort	O(n²)	O(n²)	O(n²)

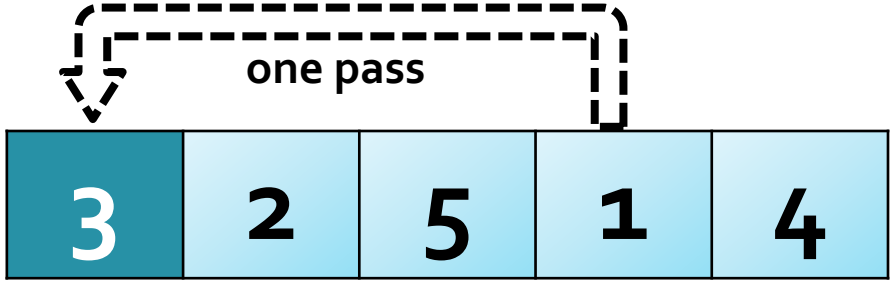
Selection Sort

ref : Youtube

Selection Sort

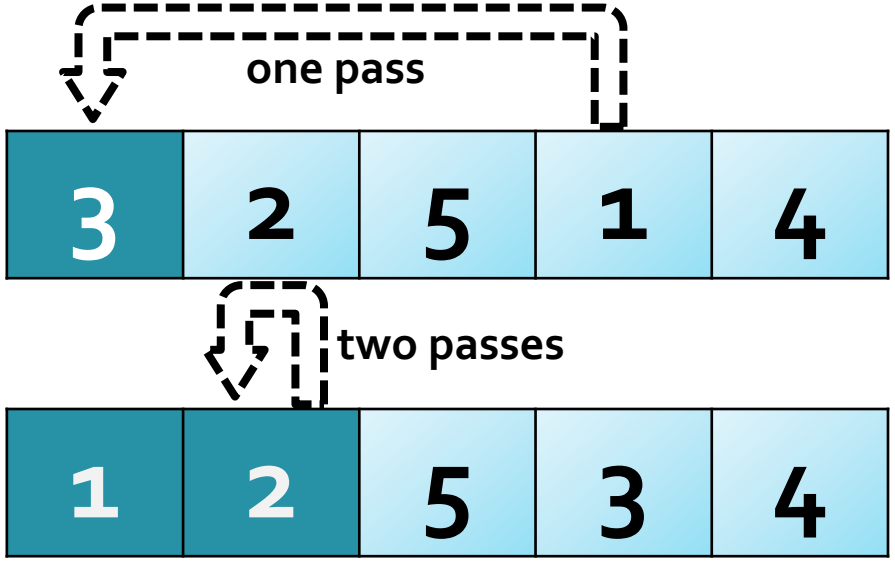
***Lets sort** this!*

Selection Sort



1 is smallest

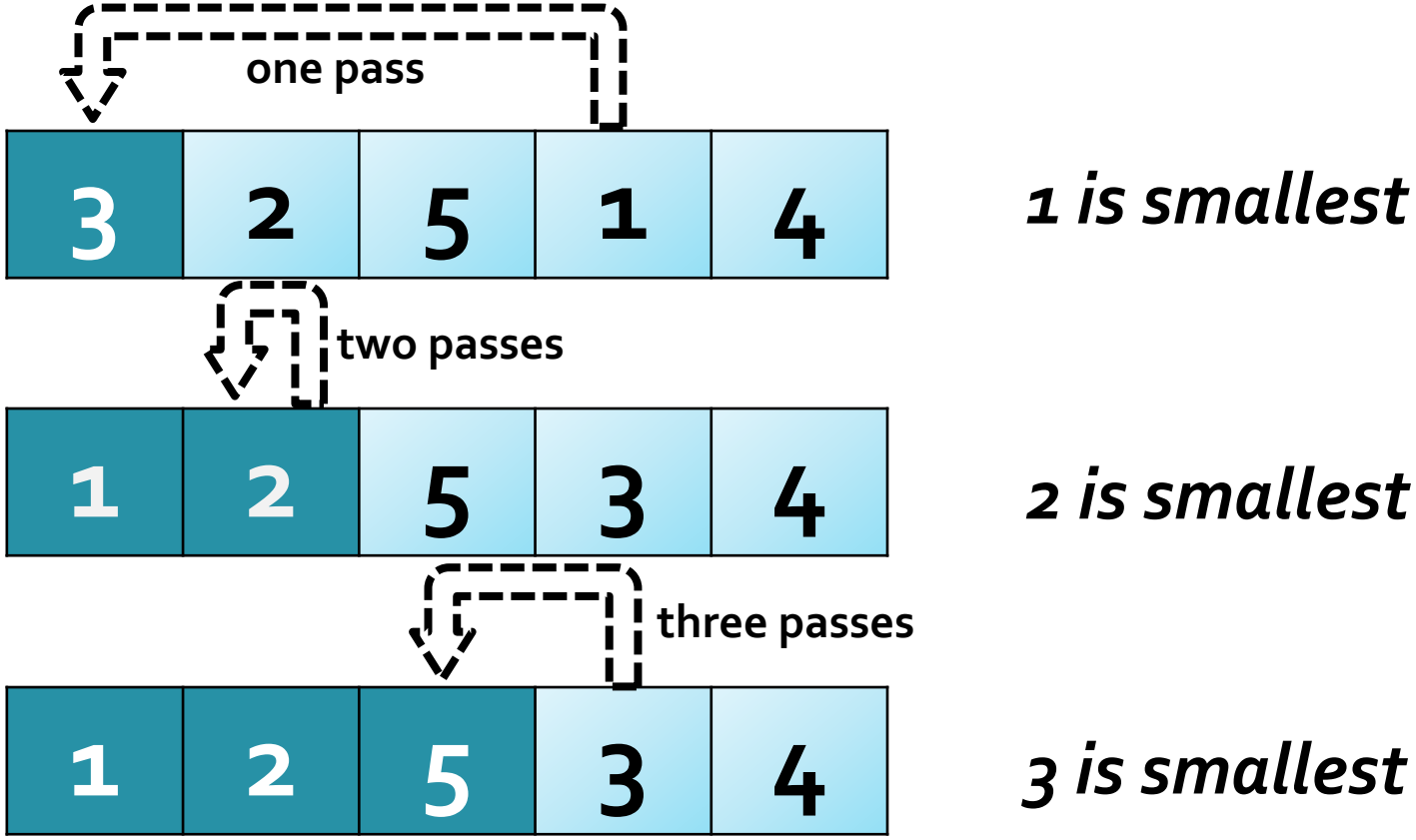
Selection Sort



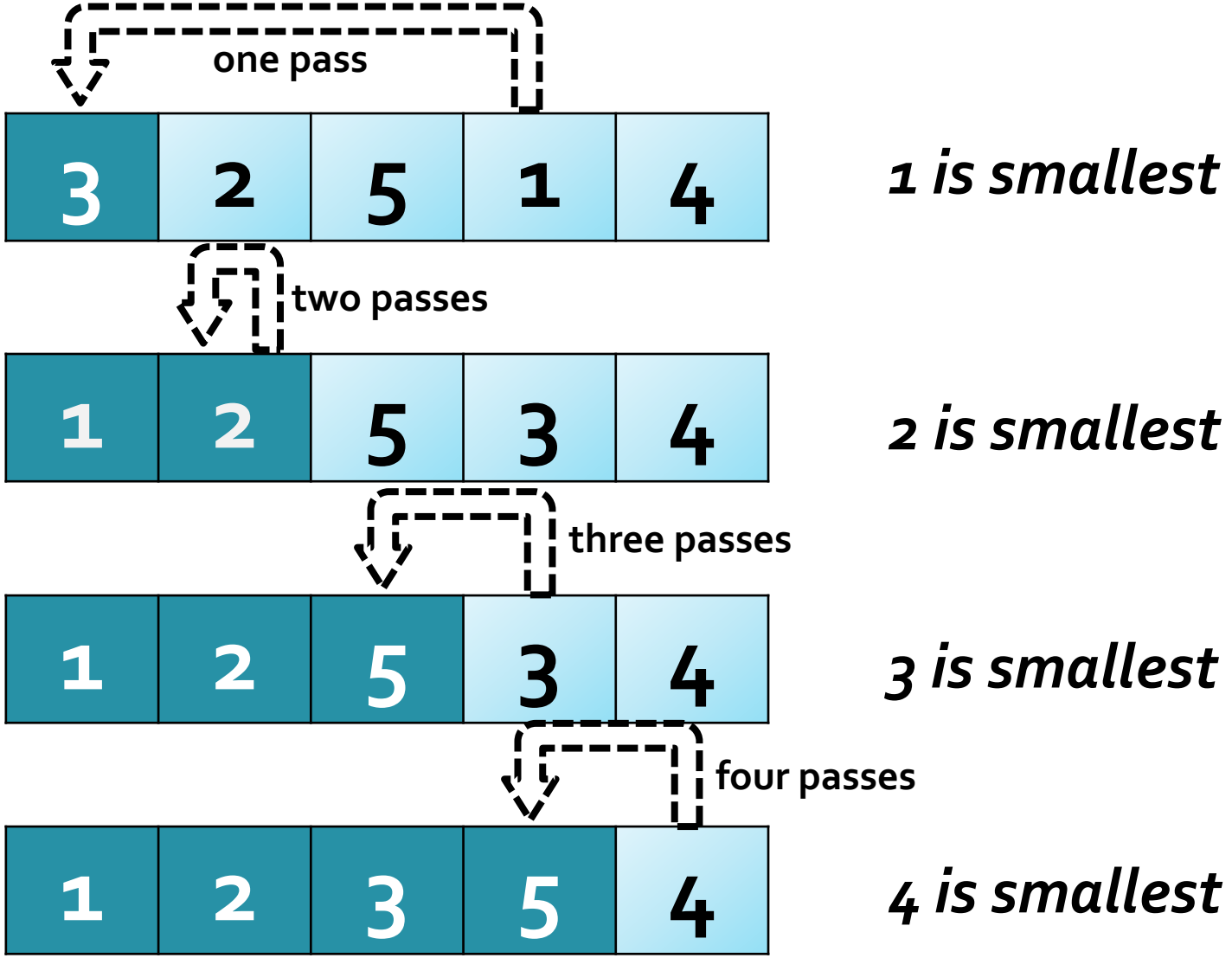
1 is smallest

2 is smallest

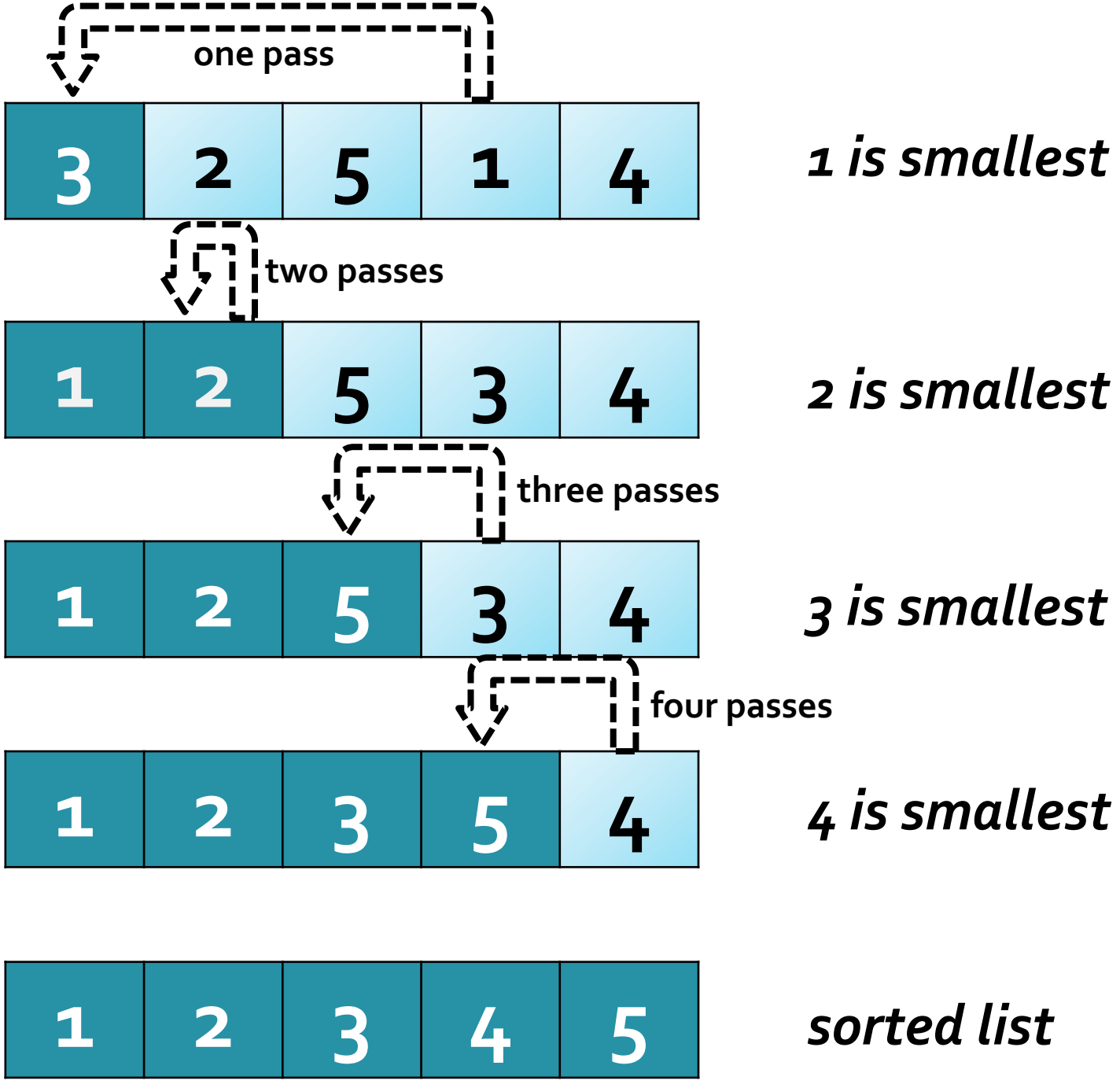
Selection Sort



Selection Sort



Selection Sort



Selection Sort

- Implementation of the **Selection sort**

```
def find_smallest(arr):
    smallest = arr[0]
    smallest_index = 0
    for i in range(1, len(arr)):
        if arr[i] < smallest:
            smallest = arr[i]
            smallest_index = i
    return smallest_index
```

Selection Sort

❖Implementation of the **Selection sort**

```
def selection_sort(arr):
    new_arr = []
    for i in range(len(arr)):
        smallest = find_smallest(arr)
        new_arr.append(arr.pop(smallest))
    return new_arr
```

```
music_count = [156, 141, 35, 94, 88, 61, 111]
music_sorted = selection_sort(music_count)

print('sorted:', music_sorted)
```

[Output]:

```
sorted:  [35, 61, 88, 94, 111, 141, 156]
```

Quiz

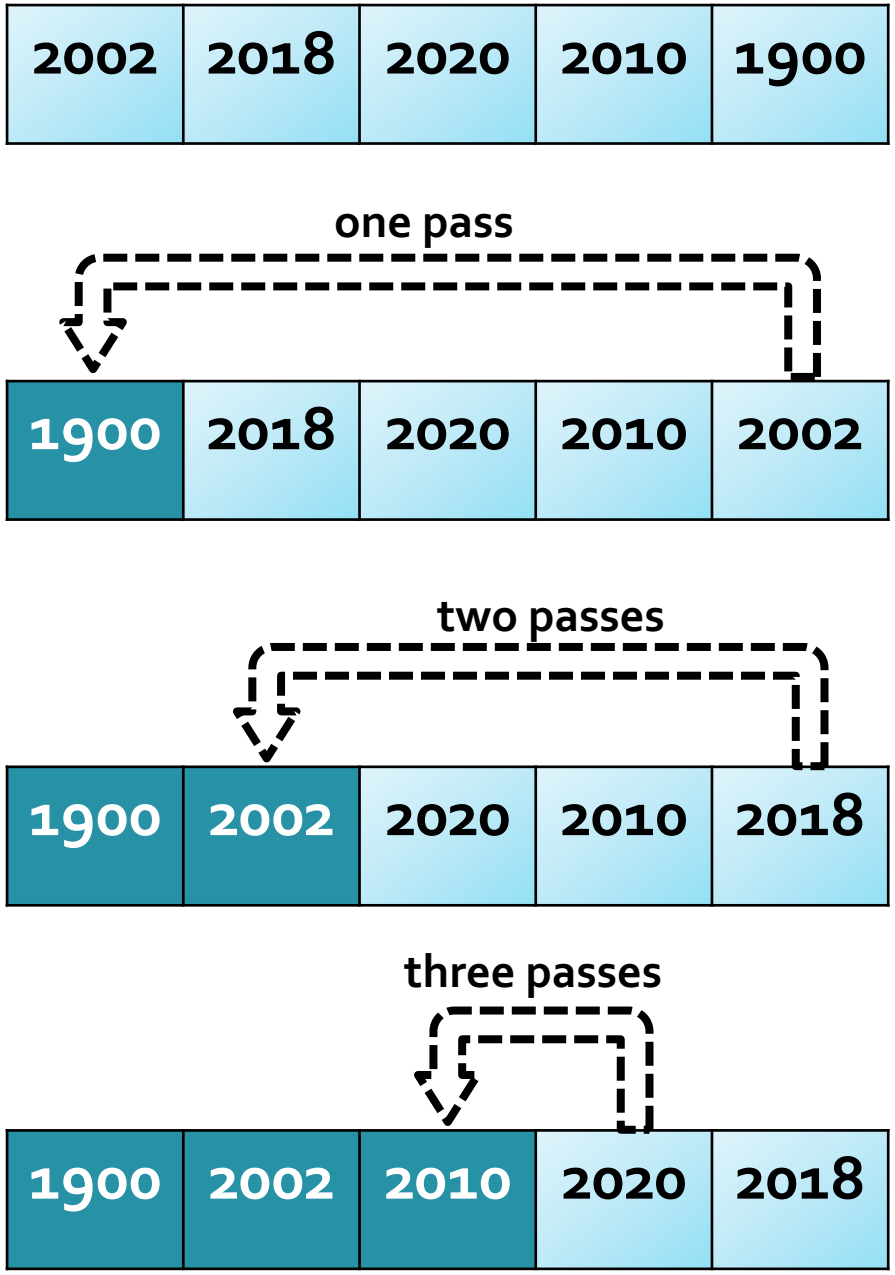
- Suppose you have a list of numbers **to sort:**

  [2002, 2018, 2020, 2010, 1900]

- which of the following lists is the partially sorted list after **three passes** of **selection sort**?

a) [2002, 2018, 2020, 2010, 1900]
b) [1900, 2018, 2020, 2010, 2002]
c) [1900, 2002, 2020, 2010, 2018]
d) [1900, 2002, 2010, 2020, 2018]
e) None of the above

Answer


2002 2018 2020 2010 1900
one pass
1900 2018 2020 2010 2002
1900 is smallest
two passes
1900 2002 2020 2010 2018
2002 is smallest
three passes
1900 2002 2010 2020 2018
2010 is smallest

Answer

- Suppose you have a list of numbers to sort:

  [2002, 2018, 2020, 2010, 1900]

- which of the following lists is the partially sorted list after **three passes** of **selection sort**?

a) [2002, 2018, 2020, 2010, 1900]
b) [1900, 2018, 2020, 2010, 2002]
c) [1900, 2002, 2020, 2010, 2018]
d) **[1900, 2002, 2010, 2020, 2018]**
e) None of the above

Insertion Sort

- Lets check another sorting algorithm called **Insertion Sort**

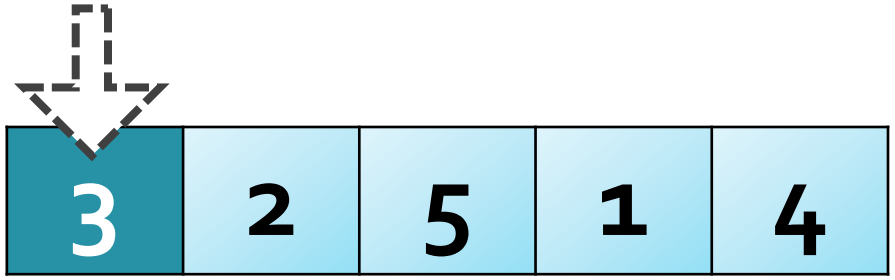
Insertion Sort

ref : Youtube

Insertion Sort

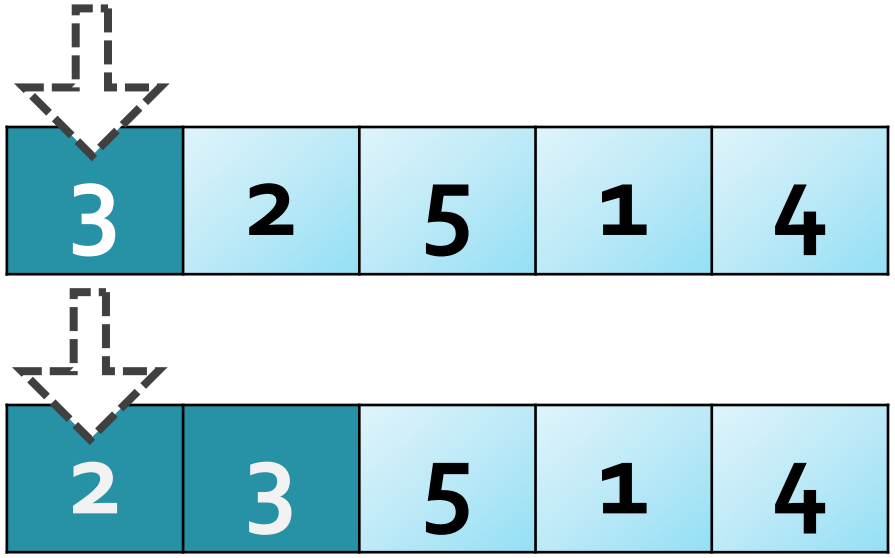
***Lets sort** this!*

Insertion Sort



3 is already sorted

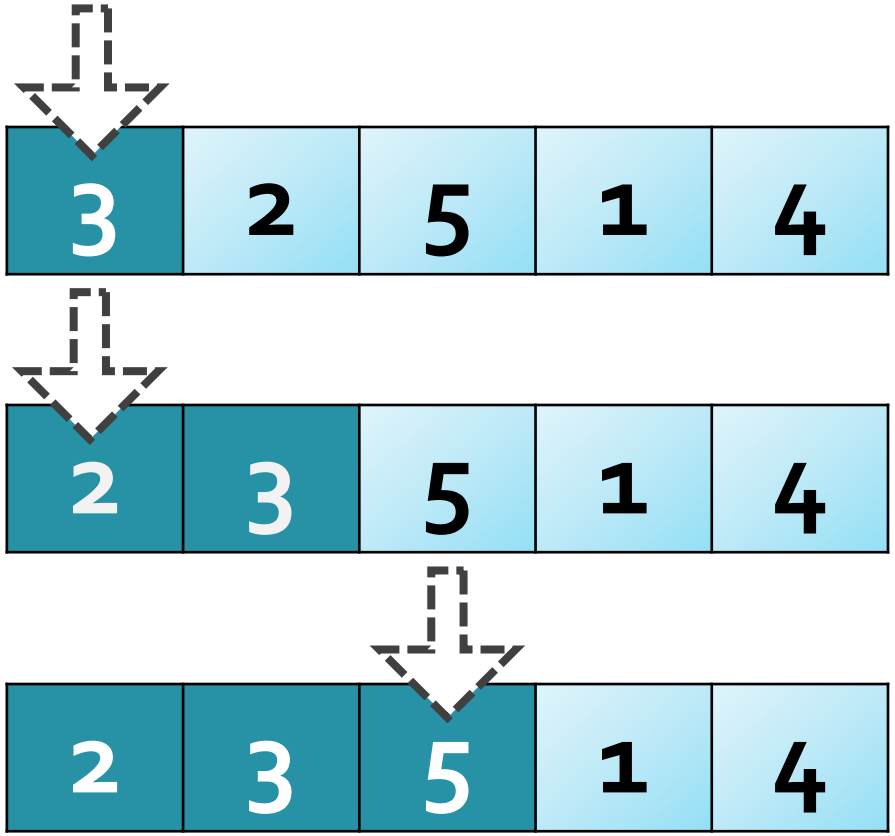
Insertion Sort



3 is alread sorted

inserted 2

Insertion Sort

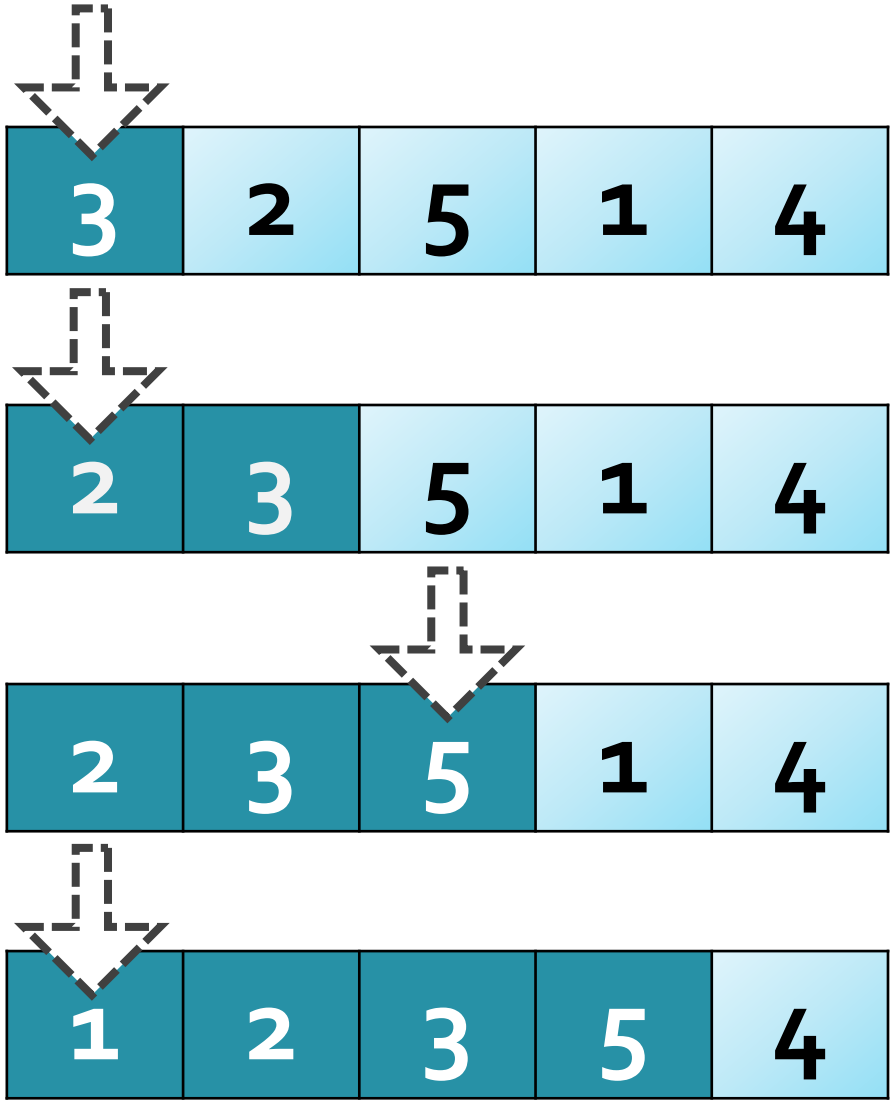


3 is alread sorted

inserted 2

inserted 5

Insertion Sort



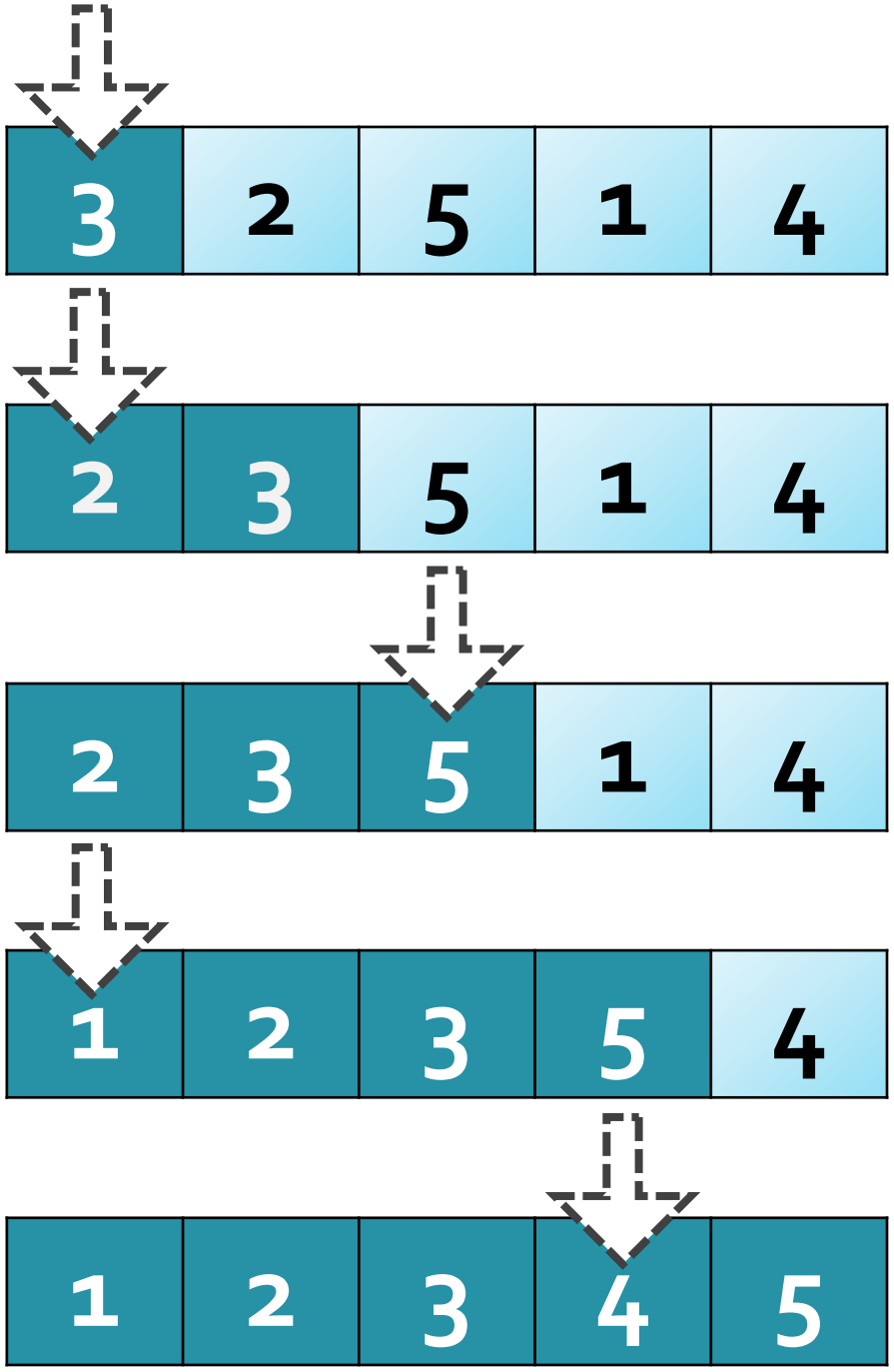
3 is alread sorted

inserted 2

inserted 5

inserted 1

Insertion Sort



3 is alread sorted

inserted 2

inserted 5

inserted 1

inserted 4

Insertion Sort

❖Implementation of the **Insertion sort**

```
def insertion_sort(arr):

    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1

        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
```

Insertion Sort

- Implementation of the **Insertion sort**

```
music_count = [156, 141, 35, 94, 88, 61, 111]
insertion_sort(music_count)
print('sorted: ', music_count)
```

[Output]:

```
sorted:  [35, 61, 88, 94, 111, 141, 156]
```

Insertion Sort vs Selection Sort

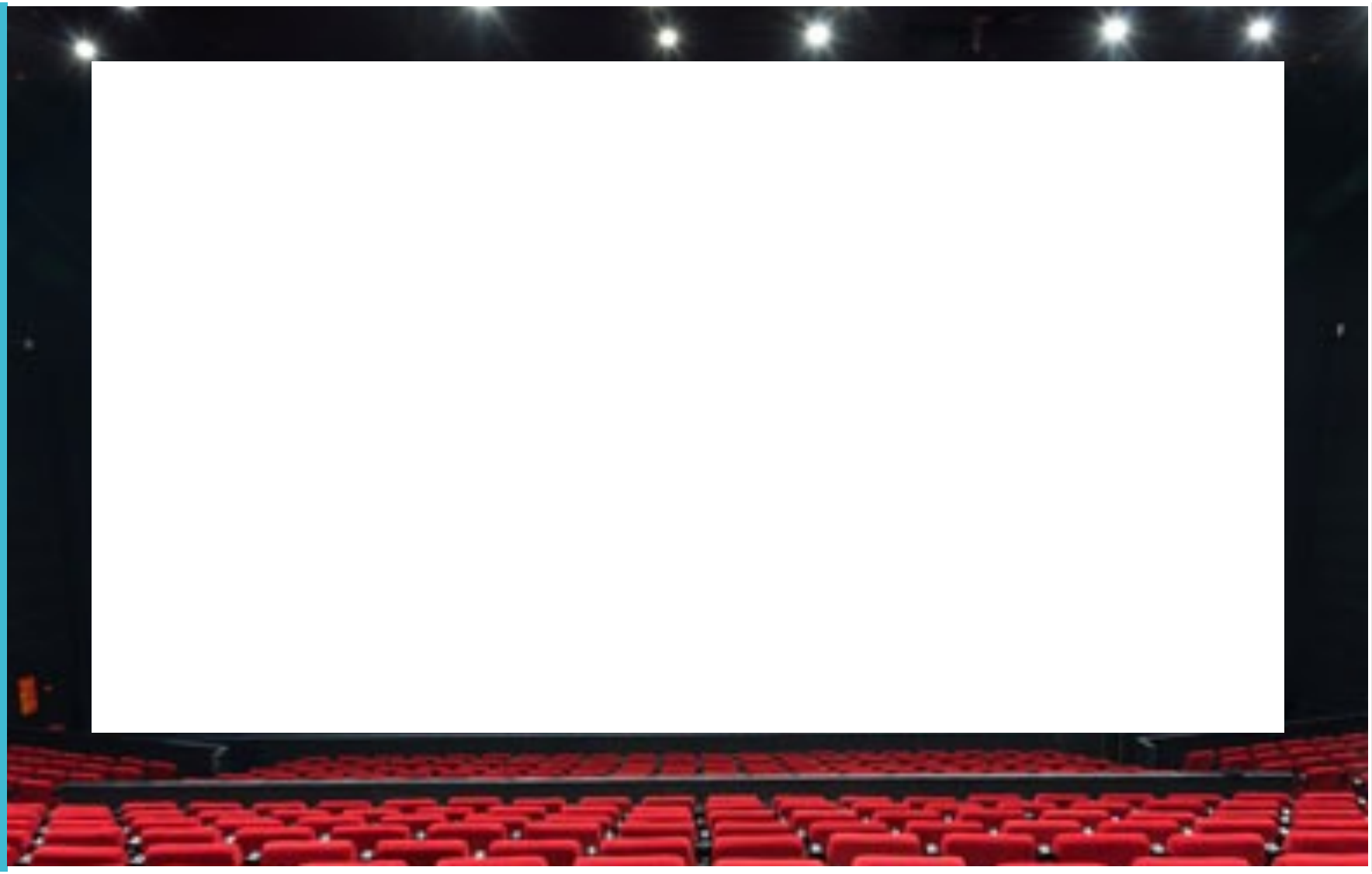
❖ **Comparison** of Insertion and Selection Sort algorithm.

	Best case	Average case	Worse case
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$

Bubble Sort

- One of the **simplest sorting** algorithms.
- It makes **multiple passes** through a list and compares **adjacent elements** and exchanges those that are out of order.
- Each pass through the list places the next largest value in its proper place.
- Each item “bubbles” up to the location where it belongs.

Bubble Sort

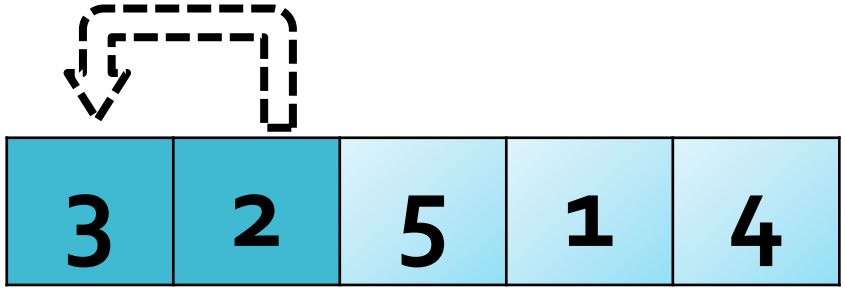


ref : Youtube

Bubble Sort

*Lets **Bubble sort** this!*

Bubble Sort



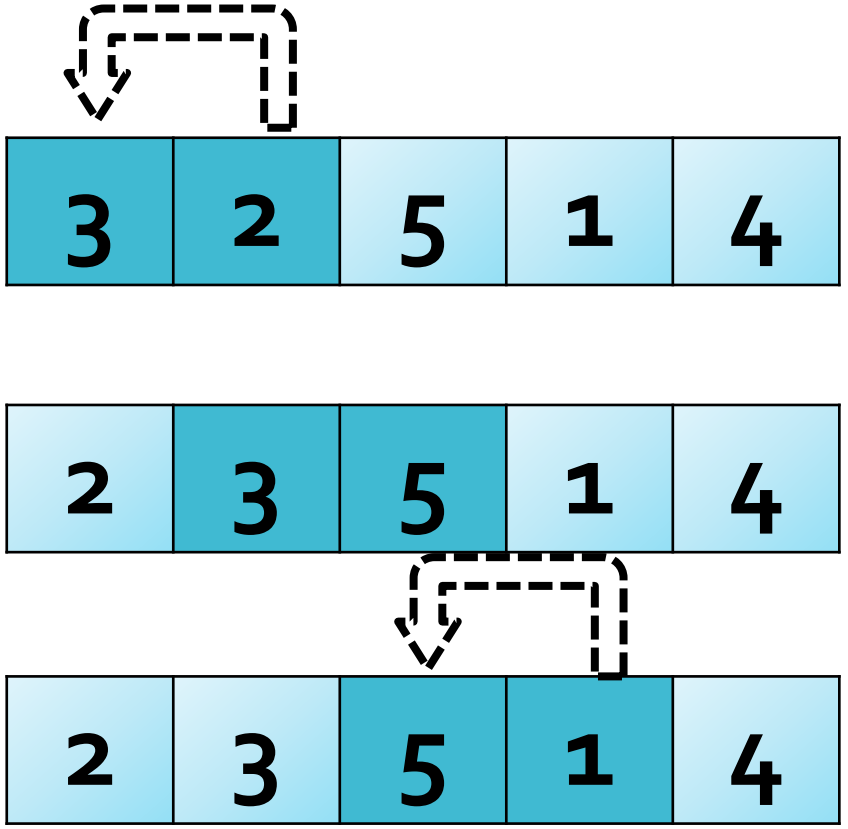
2 is smaller than 3: ***swap!***

Bubble Sort

2 is smaller than 3: ***swap!***

5 is ***not*** *smaller than 3:* ***no swap!***

Bubble Sort

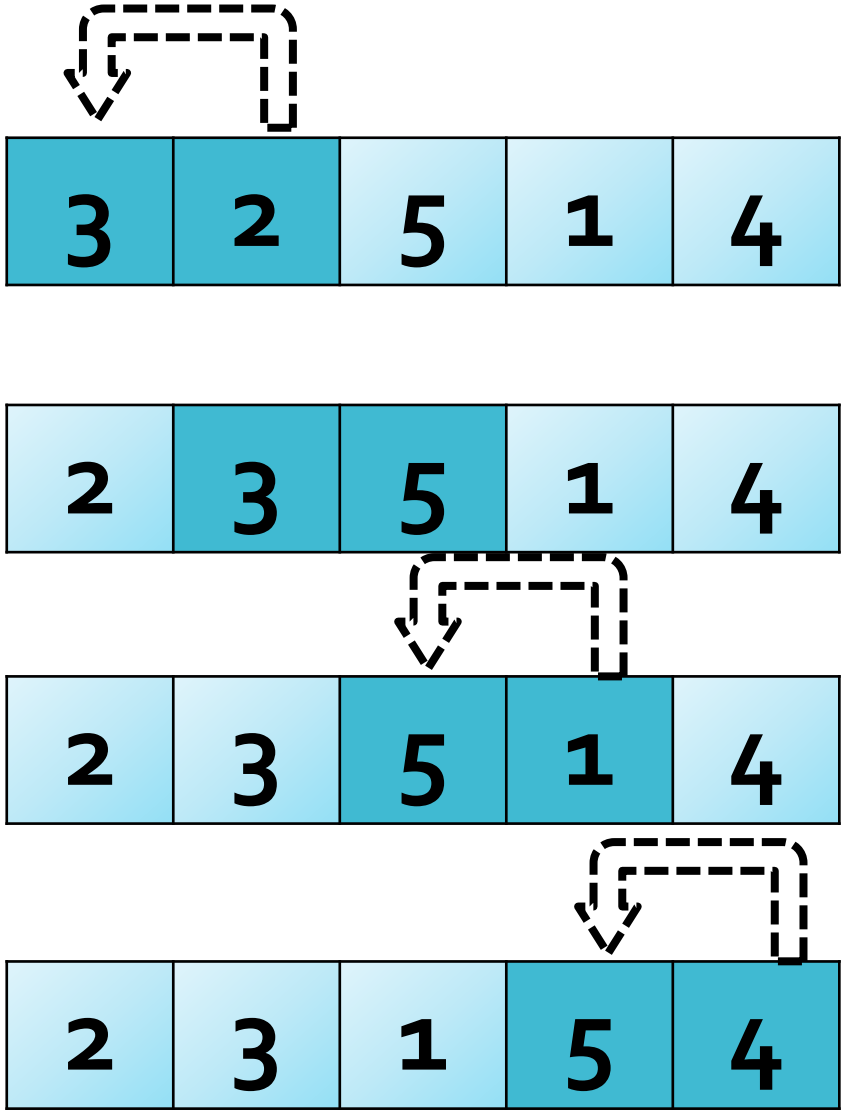


2 is smaller than 3: ***swap!***

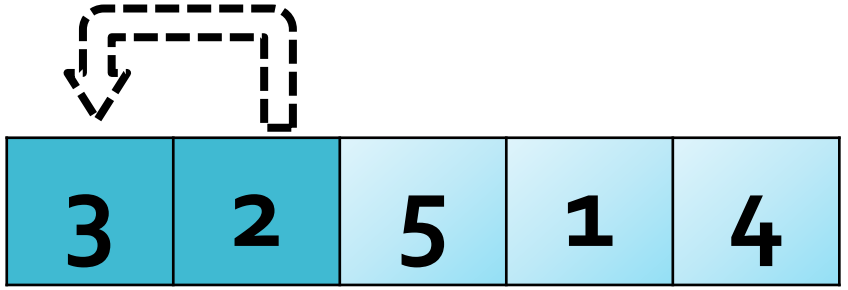
5 is ***not*** *smaller than 3:* ***no swap!***

1 is smaller than 5: ***swap!***

Bubble Sort

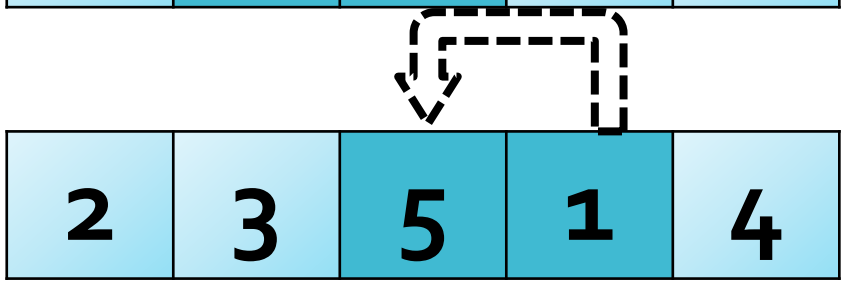

3 2 5 1 4
2 is smaller than 3: swap!
2 3 5 1 4
5 is not smaller than 3: no swap!
2 3 5 1 4
1 is smaller than 5: swap!
2 3 1 5 4
4 is smaller than 5: swap!

Bubble Sort

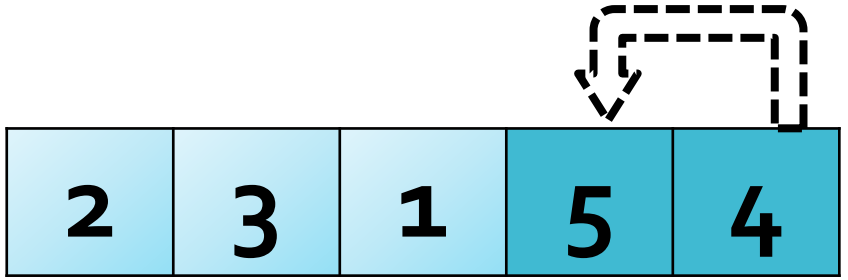


2 is smaller than 3: ***swap!***

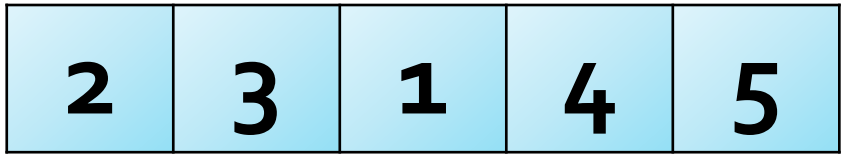
5 is ***not*** *smaller than 3:* ***no swap!***



1 is smaller than 5: ***swap!***

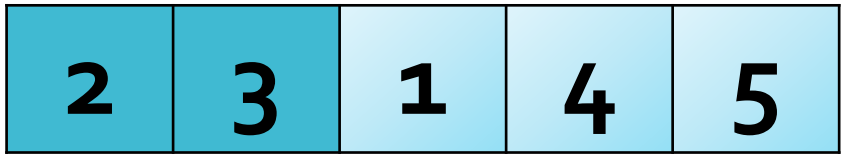


4 is smaller than 5: ***swap!***



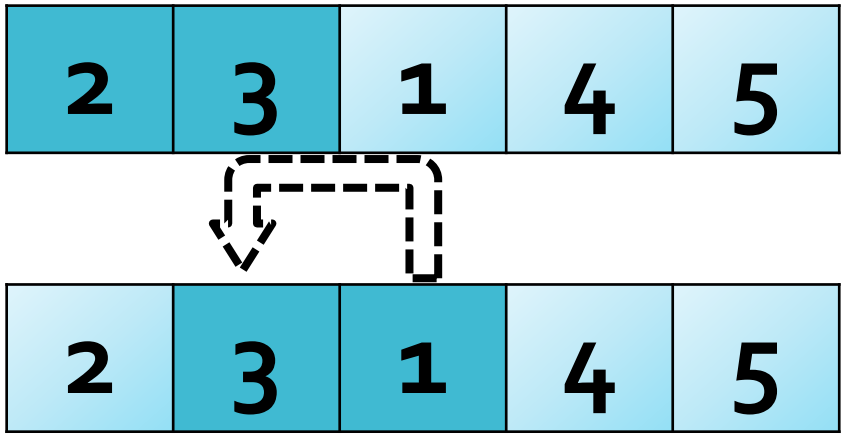
1st pass done!

Bubble Sort



*3 is **not** smaller than 2: **no swap!***

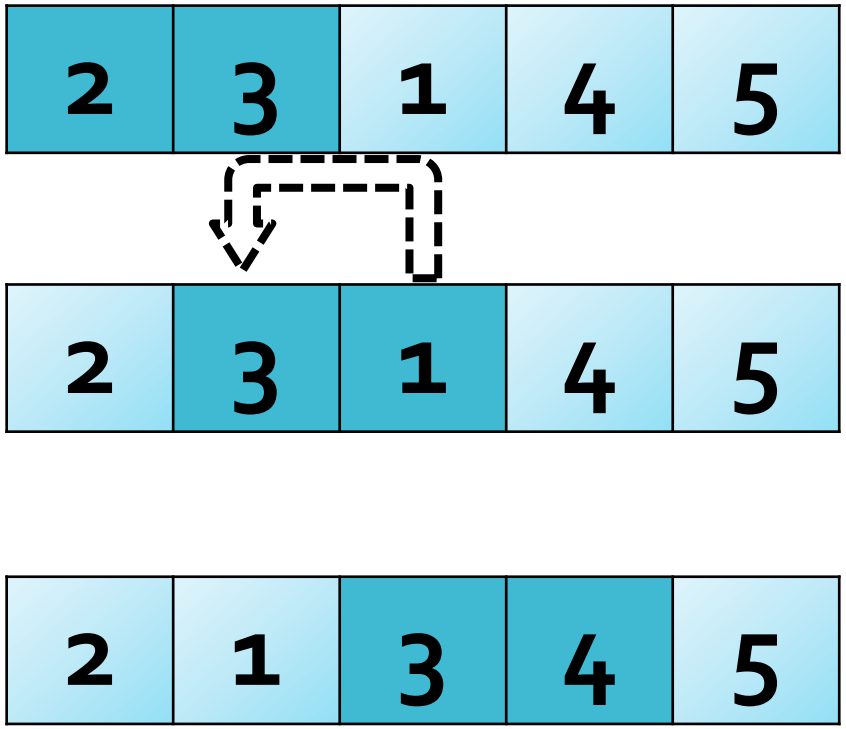
Bubble Sort



*3 is **not** smaller than 2: **no swap!***

*1 is smaller than 3: **swap!***

Bubble Sort

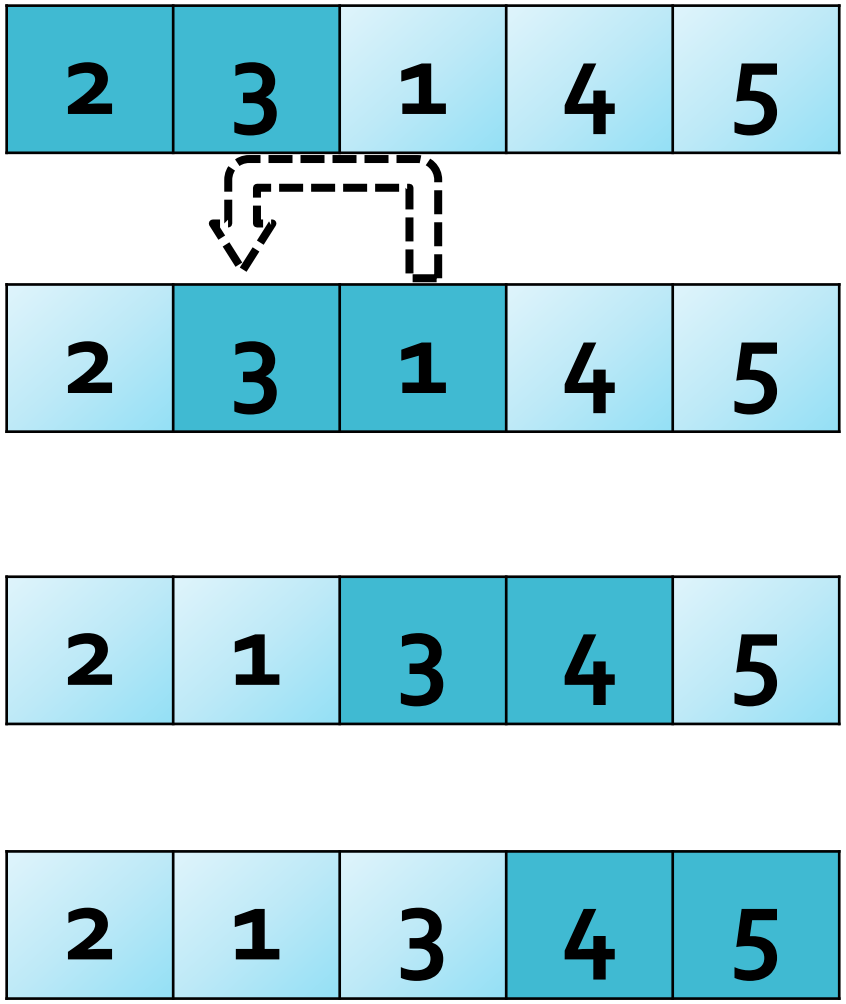


*3 is **not** smaller than 2: **no swap!***

*1 is smaller than 3: **swap!***

*4 is **not** smaller than 3: **no swap!***

Bubble Sort



*3 is **not** smaller than 2: **no swap!***

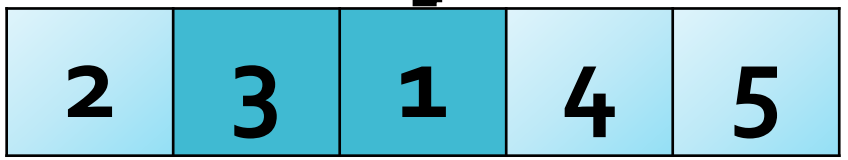
*1 is smaller than 3: **swap!***

*4 is **not** smaller than 3: **no swap!***

*5 is **not** smaller than 4: **no swap!***

Bubble Sort

*3 is **not** smaller than 2: **no swap!***



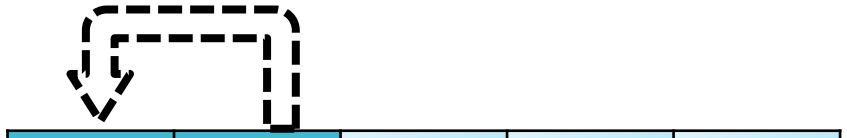
*1 is smaller than 3: **swap!***

*4 is **not** smaller than 3: **no swap!***

*4 is **not** smaller than 3: **no swap!***

2nd pass done!

Bubble Sort



2	1	3	4	5

1 is smaller than 2: ***swap!***

1	2	3	4	5

3 is ***not*** *smaller than 2:* ***no swap!***

1	2	3	4	5

4 is ***not*** *smaller than 3:* ***no swap!***

1	2	3	4	5

5 is ***not*** *smaller than 4:* ***no swap!***

1	2	3	4	5

3rd pass done!

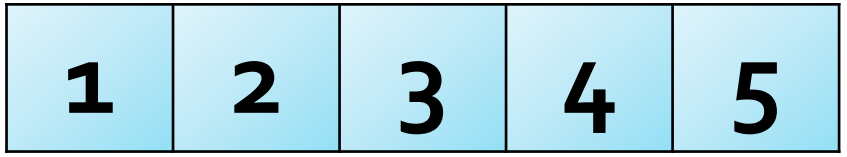
Bubble Sort

*2 is **not** smaller than 2: **swap!***

*3 is **not** smaller than 2: **no swap!***

*4 is **not** smaller than 3: **no swap!***

*5 is **not** smaller than 4: **no swap!***



4th pass done!

Bubble Sort

❖Implementation of the **Bubble sort**

```
def bubble_sort(arr):

    for pass_num in range(len(arr) - 1, 0, -1):

        for i in range(pass_num):
            if arr[i] > arr[i + 1]:
                temp = arr[i]
                arr[i] = arr[i + 1]
                arr[i + 1] = temp
```

```
music_count = [156, 141, 35, 94, 88, 61, 111]
bubble_sort(music_count)

print(music_count)
```

[Output] sorted: [35, 61, 88, 94, 111, 141, 156]

Bubble Sort

	Best case	Average case	Worse case
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$

Quiz

- This is IMDB Top most movies in 2020.
- You are given a **List of Tuples**, each representing the (**title**, **rating**) of movies.
- Write a Python function that uses **Bubble Sort** and sorts movies by the ratings.

ratings

[Input]:

```
list_of_tuples = [('Birds of Prey', 6.6),
                  ('Dolittle', 5.5),
                  ('The Gentlemen', 8.1),
                  ('Falling', 7.5),
                  ('Bad Boys for Life', 7.2)]
```

Quiz

- Write a Python function that uses **Bubble Sort** and sorts movies by the ratings.

```
sorted_list = sort_list_of_tuple(list_of_tuples)
print(sorted_list)
```

[Output]:

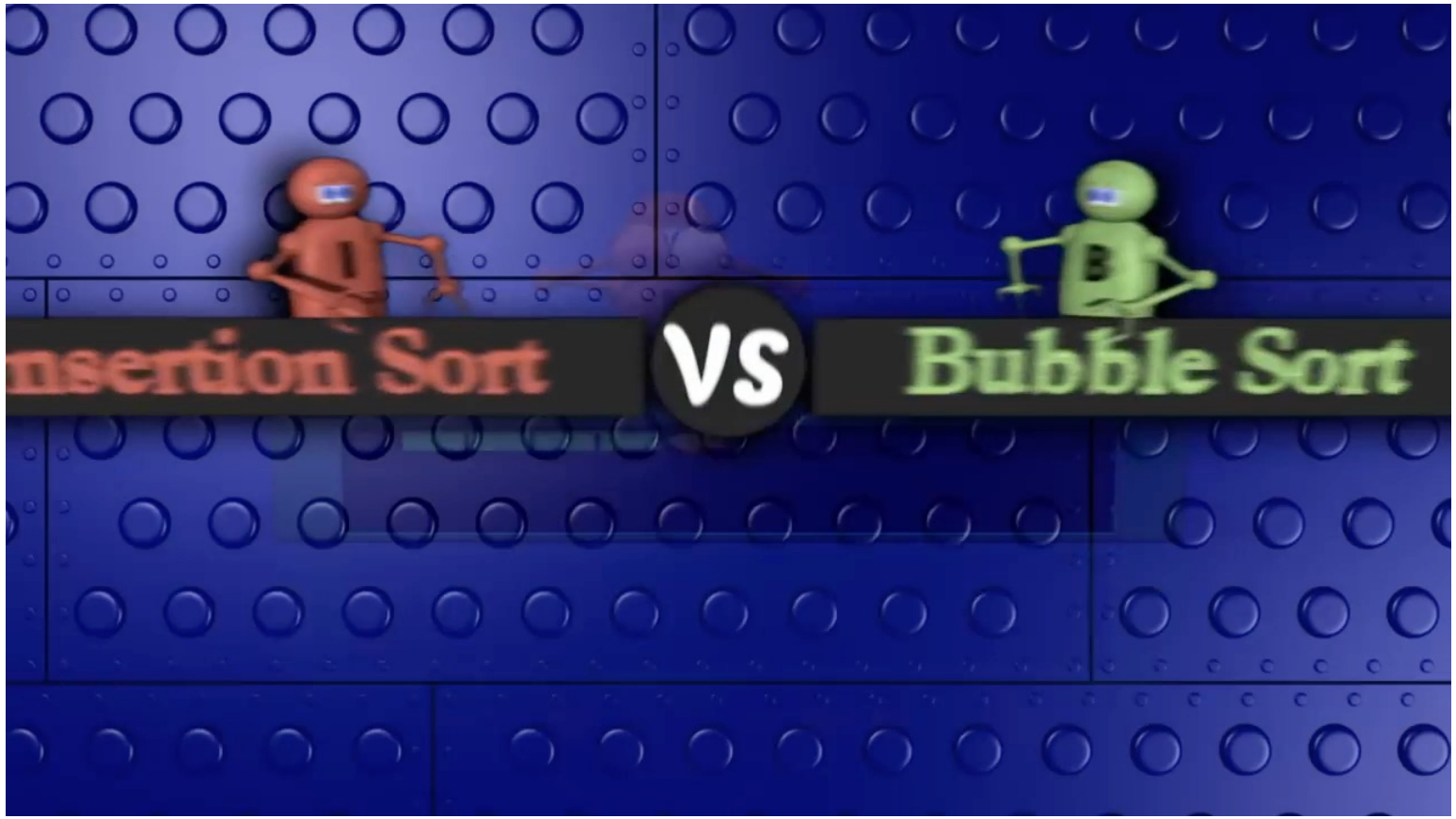
```
[('Dolittle', 5.5),
 ('Birds of Prey', 6.6),
 ('Bad Boys for Life', 7.2),
 ('Falling', 7.5),
 ('The Gentlemen', 8.1)]
```

Answer

- Write a Python function that uses **Bubble Sort** and sorts movies by the ratings.

```
def sort_list_of_tuple(list_of_tup):
    size = len(list_of_tup)
    for i in range(0, size):

        for j in range(0, size - i - 1):
            if list_of_tup[j][1] > list_of_tup[j + 1][1]:
                temp = list_of_tup[j]
                list_of_tup[j] = list_of_tup[j + 1]
                list_of_tup[j + 1] = temp
    return list_of_tup
```

Comparison



Next Lesson

- **Sorting algorithms** in Python (part2)