Object-oriented programming

Kylie A. Bemis

Northeastern University
Khoury College of Computer Sciences

Northeastern University

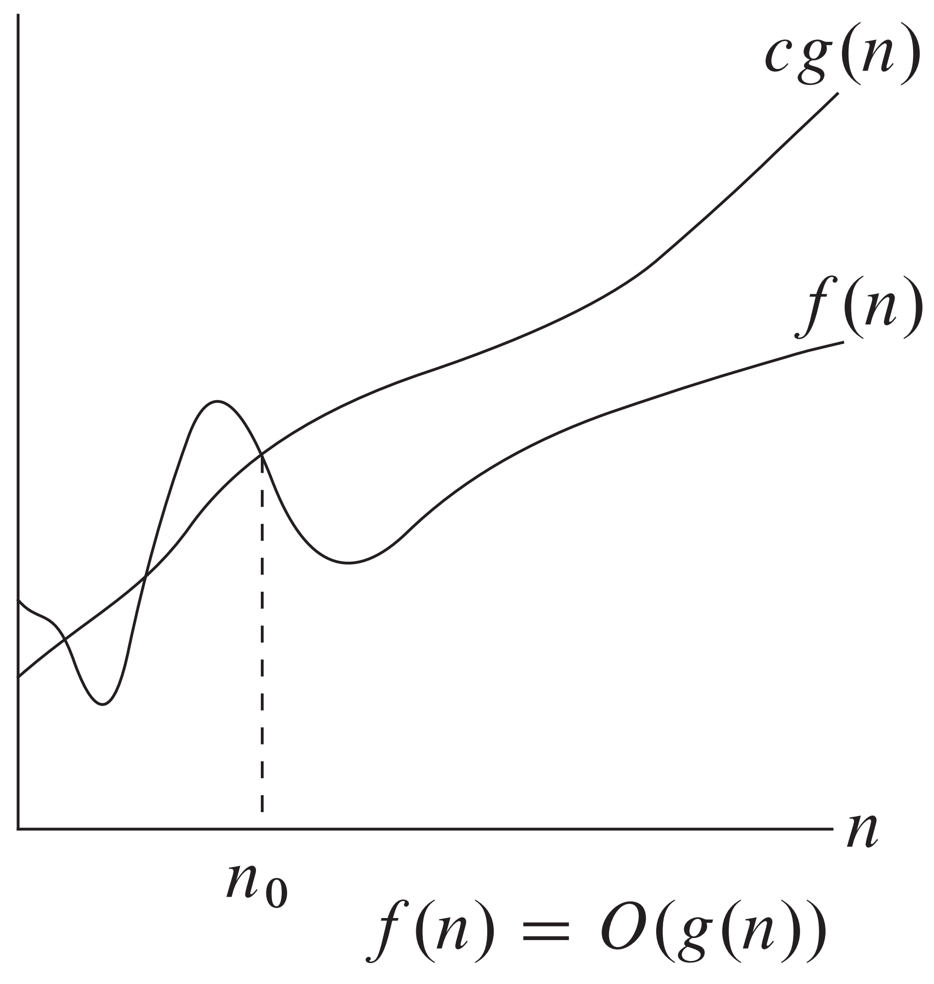
Goals for today

- Review of algorithms & complexity
- Object-oriented programming (OOP)
- Classes and methods in Python

REVIEW:
ALGORITHMS & COMPLEXITY

Measuring algorithmic complexity

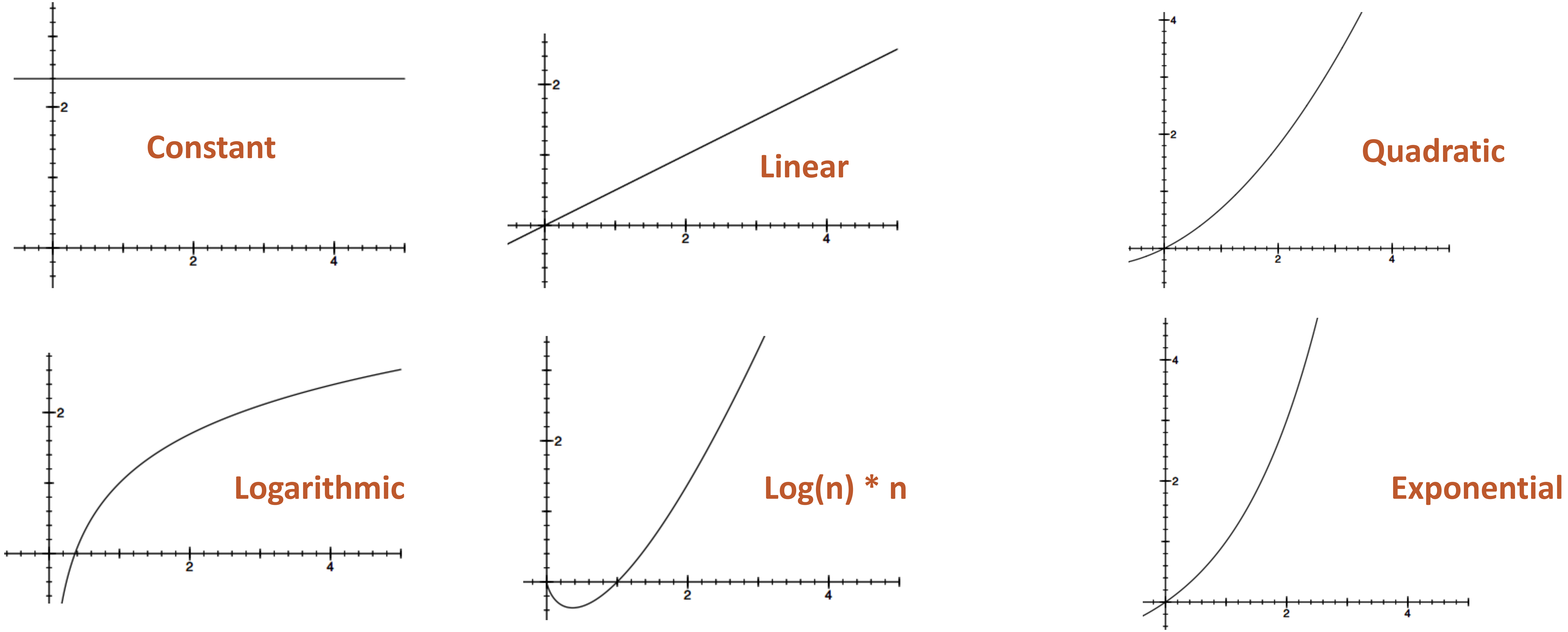
- *Upper bound* on amount of time for an algorithm to complete for input size ***n***
- Asymptotic *upper bound* described as $O(g(n))$
- A function $f(n)$ is $O(g(n))$ if
  - For some constant c and all values of $n \geq$ some value n_0
  - $f(n) \leq c \times g(n)$



E.g., $4n + 3 \rightarrow O(n)$

E.g., $4n^2 + 3 \rightarrow O(n^2)$

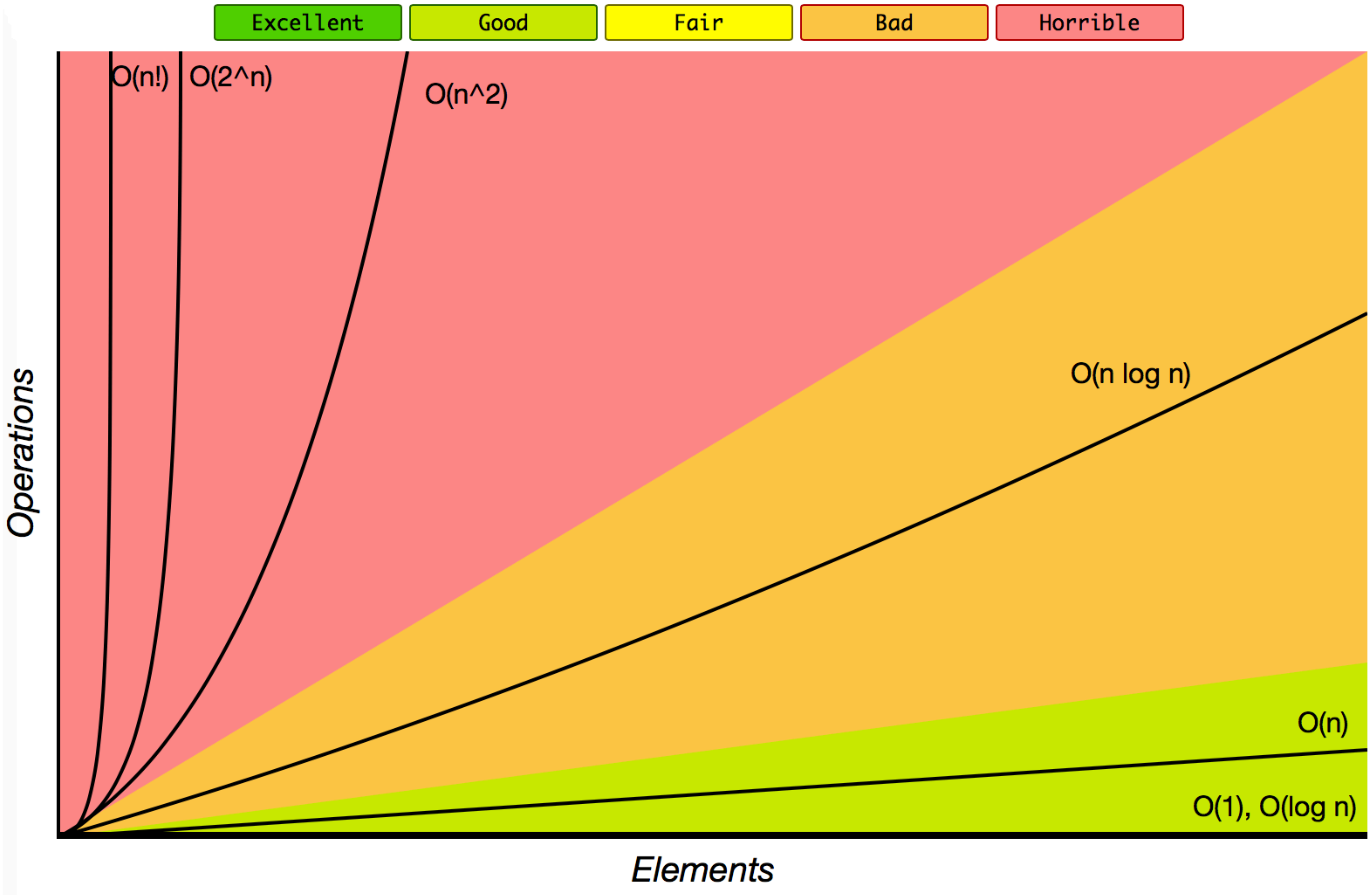
Growth rates of functions



Common complexities

O(1)	Constant time (not affected by input size)
O(n)	Time increases linearly with input
O(n^2)	Time increases quadratically with input
O(log n)	Time increases logarithmically (divide & conquer)
O(n log n)	Time increases log-linearly (divide & conquer)
O(x^n)	Time increases by factor of x for each new input
O(n!)	Time increases by a larger factor for each new input

Visualizing complexities



https://learntocodetogether.com/big-o-cheat-sheet-for-common-data-structures-and-algorithms/

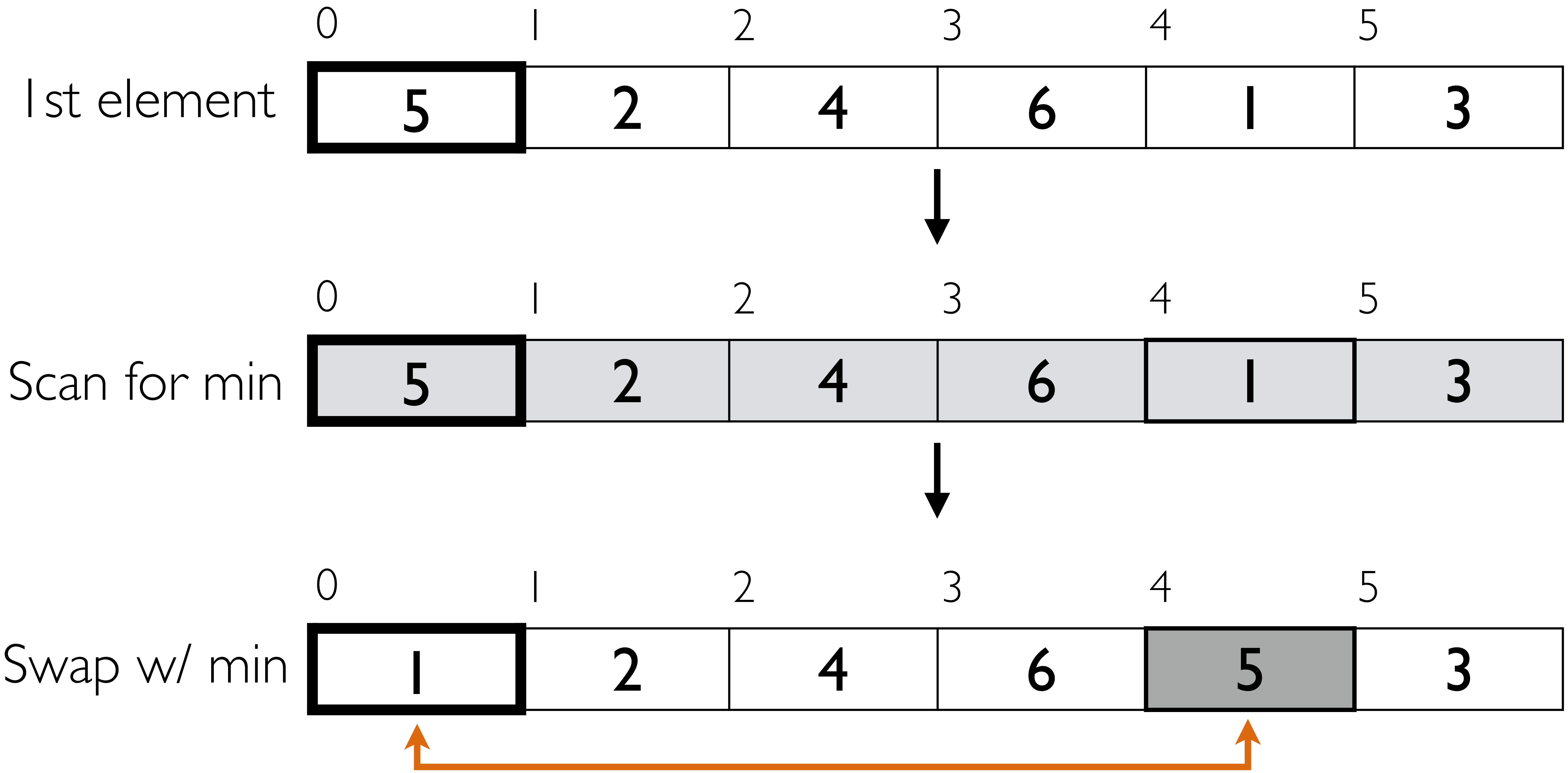
Sort an array in increasing order?

	0	1	2	3	4	5
Input	5	2	4	6	1	3

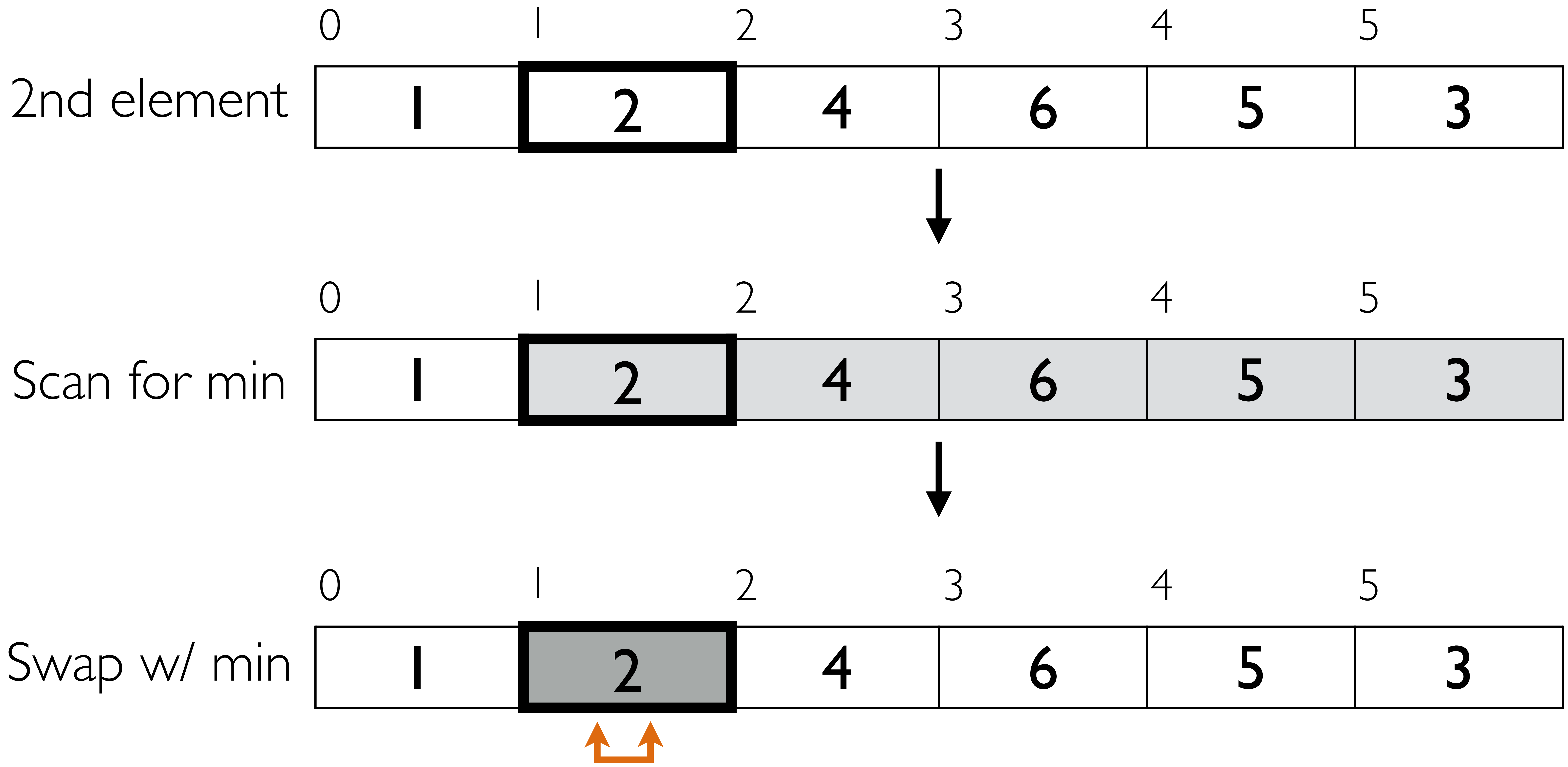
↓

	0	1	2	3	4	5
Output	1	2	3	4	5	6

1. Select the smallest element



2. Select the second smallest element



3. Select the third element

	0	1	2	3	4	5
3rd element	1	2	**4**	6	5	3
Scan for min	1	2	**4**	6	5	**3**
Swap w/ min	1	2	**3**	6	5	4

Selection sort

- For each position *i* along the array:
  - Scan forward from *i* to find smallest element
  - Swap smallest element into position *I*
  - First *i* elements are now a sorted subarray

```
SelectionSort(x):

for i along x
    find index j of min of x[i:]
    swap x[j] and x[j]
```

Complexity of selection sort

```
def ssort(x):
      """
      Sorts a list of numbers in-place using selection sort
      param x: The list to sort (in place)
      returns: None
      """
      for i in range(len(x)):
        imin = i
        # find minimum in sublist x[i:]
        for j in range(i, len(x)):
          if x[j] < x[imin]:
            imin = j
        swap(x, i, imin)
```

Outer loop grows as O(n)

↓

Inner loop grows as O(n)

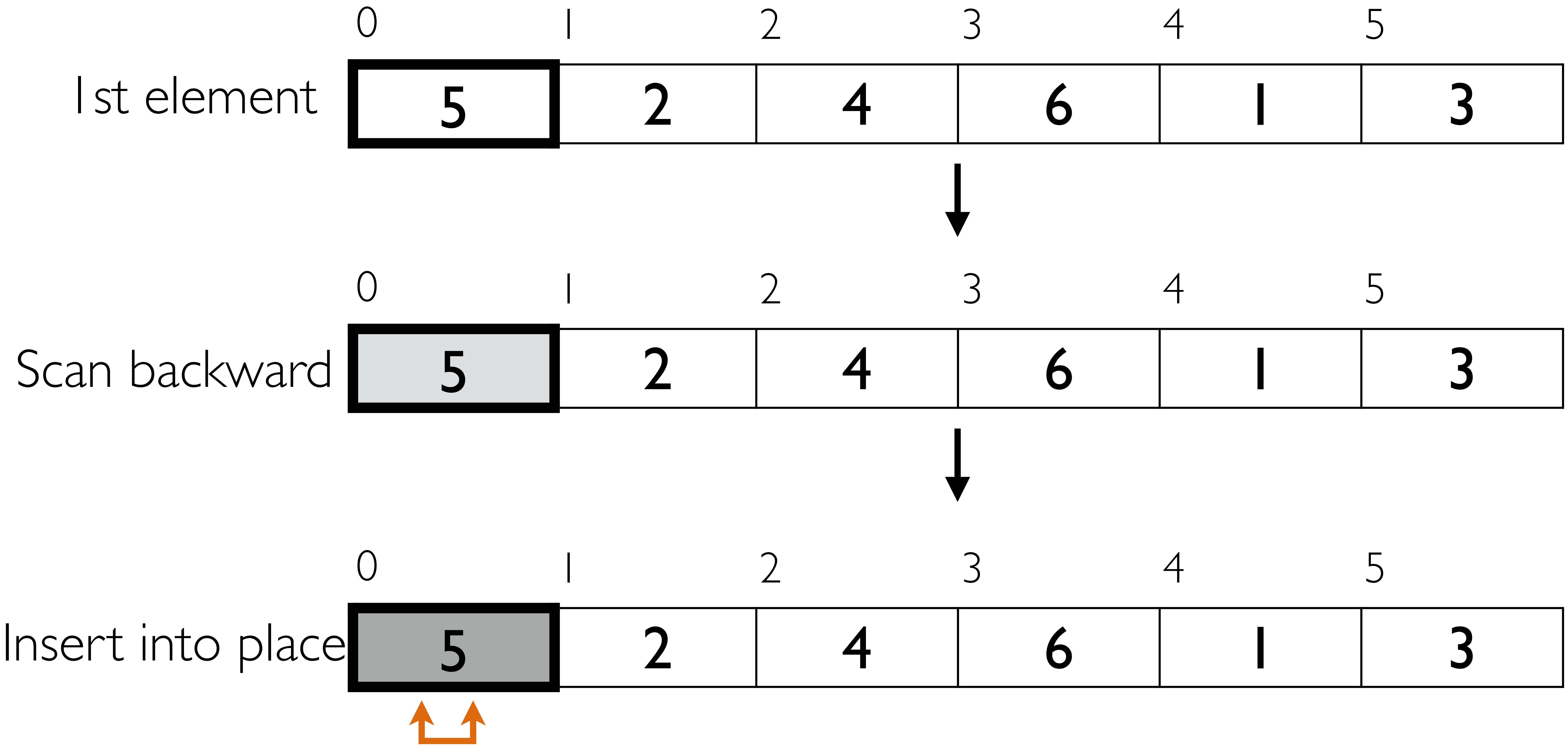
↓

$\boxed{\text{Therefore, selection sort is } O(n \times n) = O(n^2)}$

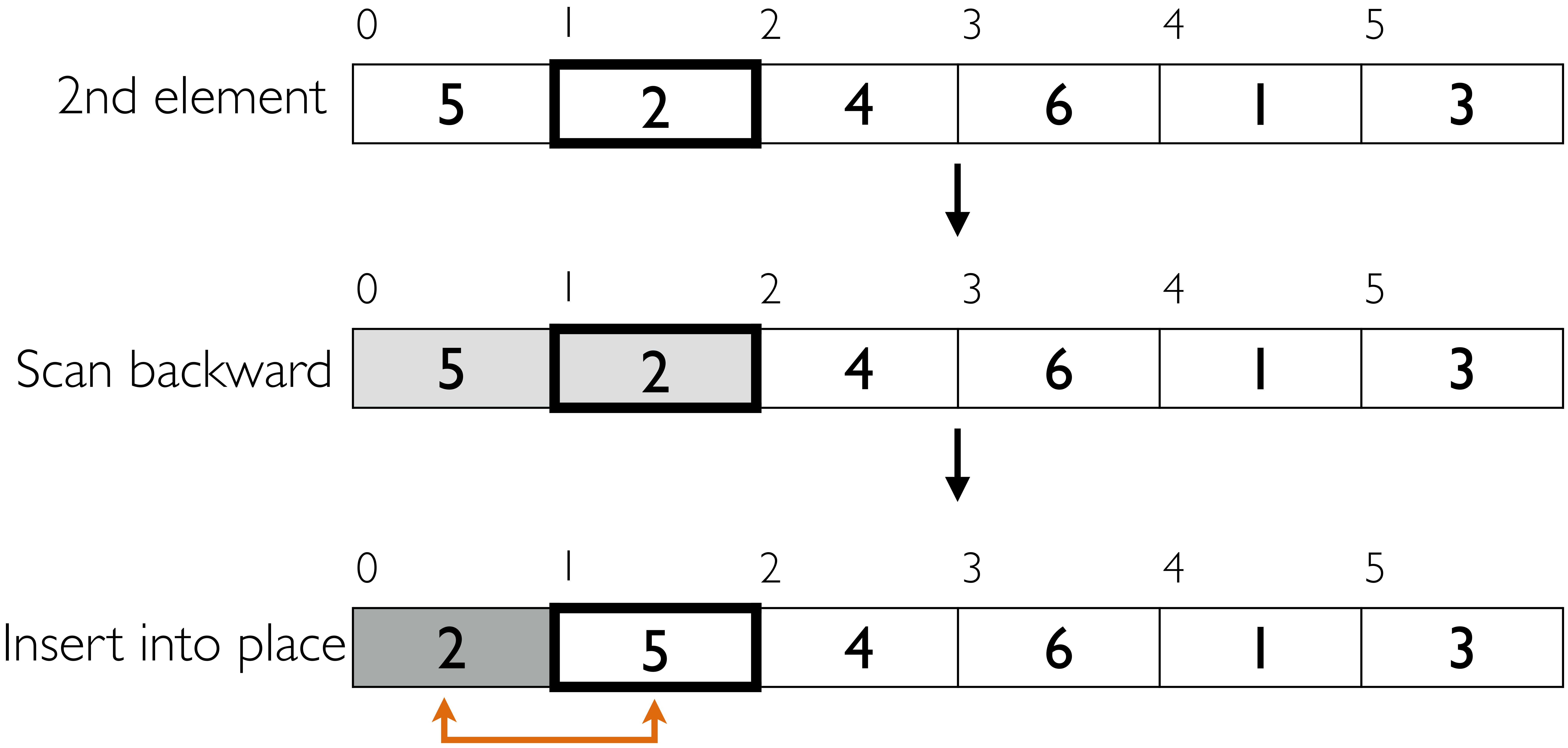
Improving selection sort

- Always need to scan whole subarray
- Is there a way we can improve this?
- Scan backward instead of forward?

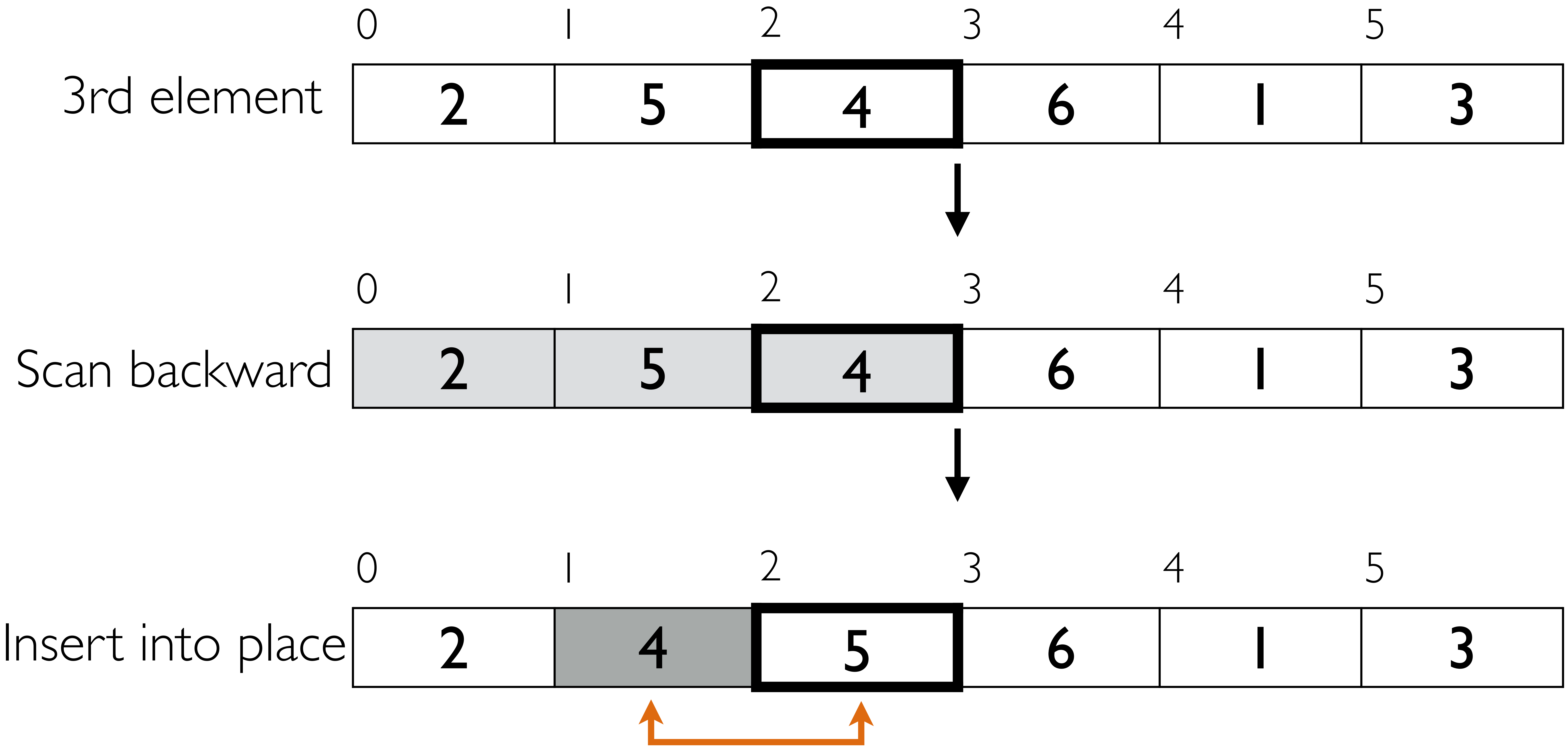
1. Insert first element into subarray



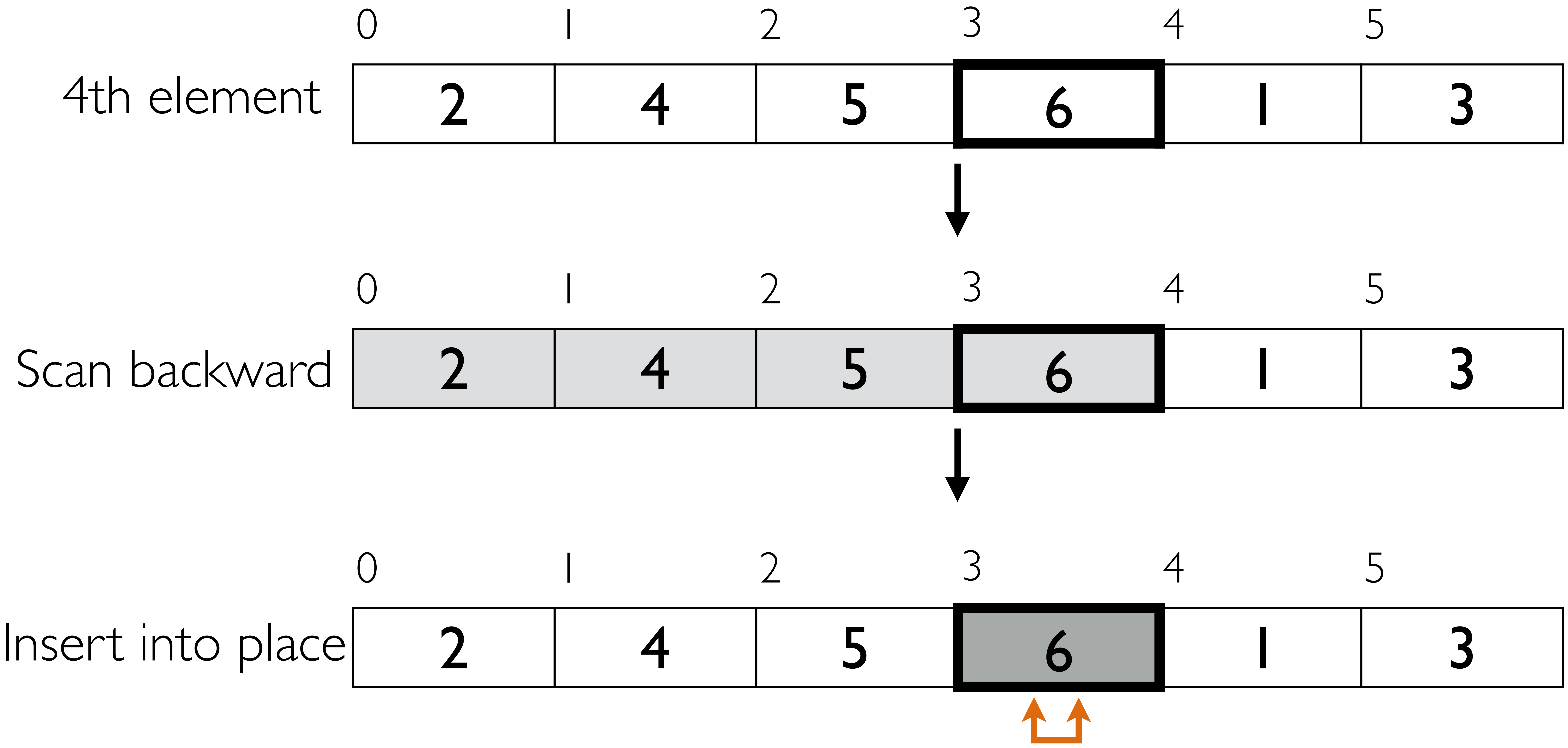
2. Insert second element into subarray



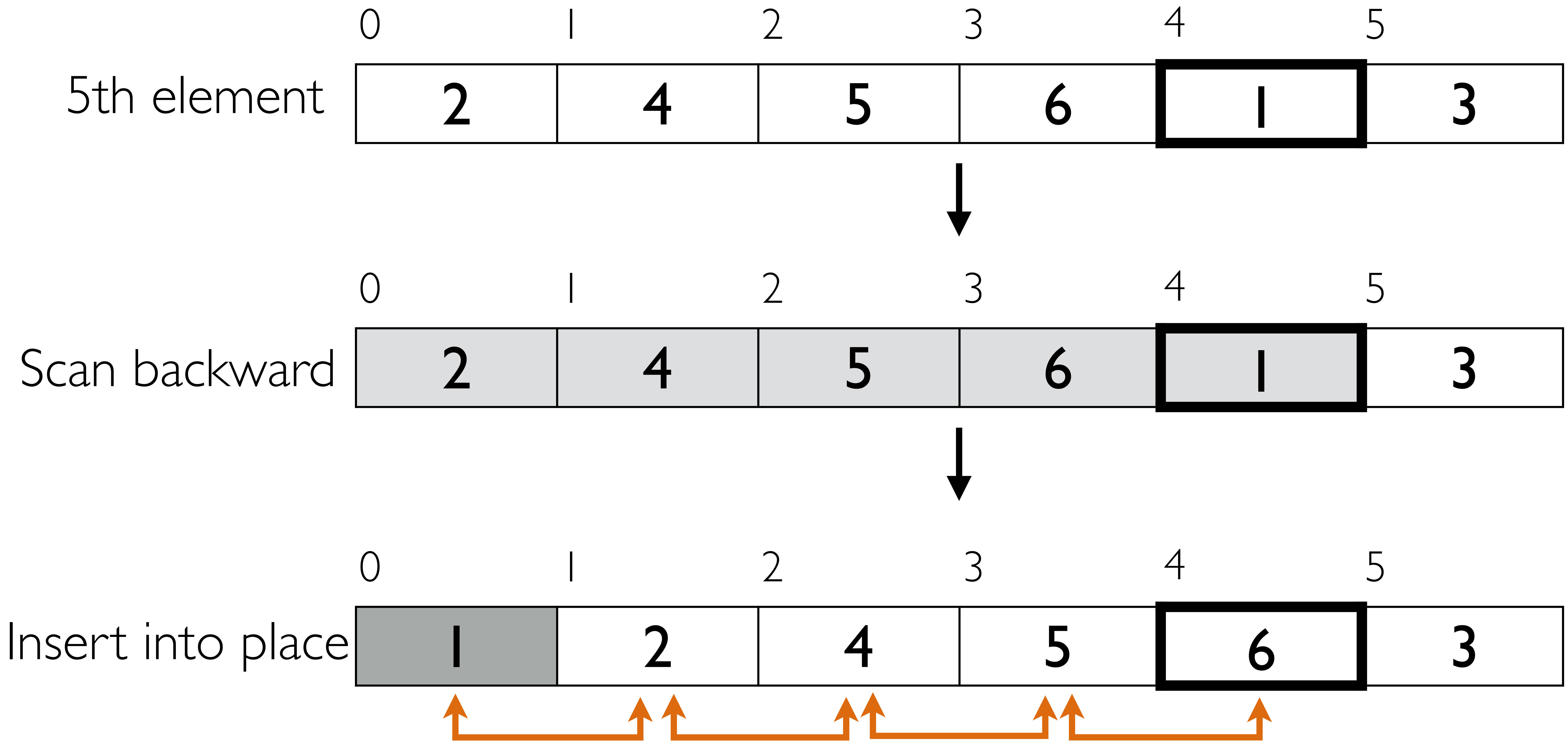
3. Insert third element into subarray



4. Insert fourth element into subarray



5. Insert fifth element into subarray



Insertion sort

- For each position *i* along the array:
  - Scan backward from *i* comparing each element
  - Swap current element toward front while it's smaller
  - First *i* elements are now a sorted subarray

```
InsertionSort(x):

for i along x
    while x[i] < x[i-1]
        swap x[i] and x[i-1]
        i = i - 1
```

Complexity of insertion sort

```
def isort(x):
    """
    Sorts a list of numbers in-place using insertion sort
    param x: The list to sort (in place)
    returns: None
    """
    for i in range(len(x)):
      j = i - 1
      # swap x[i] toward front of list
      while j >= 0 and x[i] < x[j]:
         if x[i] < x[j]:
            swap(x, i, j)
            i = j # no effect next iteration!
         j -= 1
```

Outer loop grows as O(n)

↓

Inner loop grows as O(n)

↓

Therefore, insertion sort is also $O(n \times n) = O(n^2)$

Sort array with divide and conquer?

Input

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

Output

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

Divide the problem into sub-problems

Merge the results back together

Merge sort

- Split input array into two subarrays
  - Apply merge sort to each subarray
  - Merge the sorted subarrays

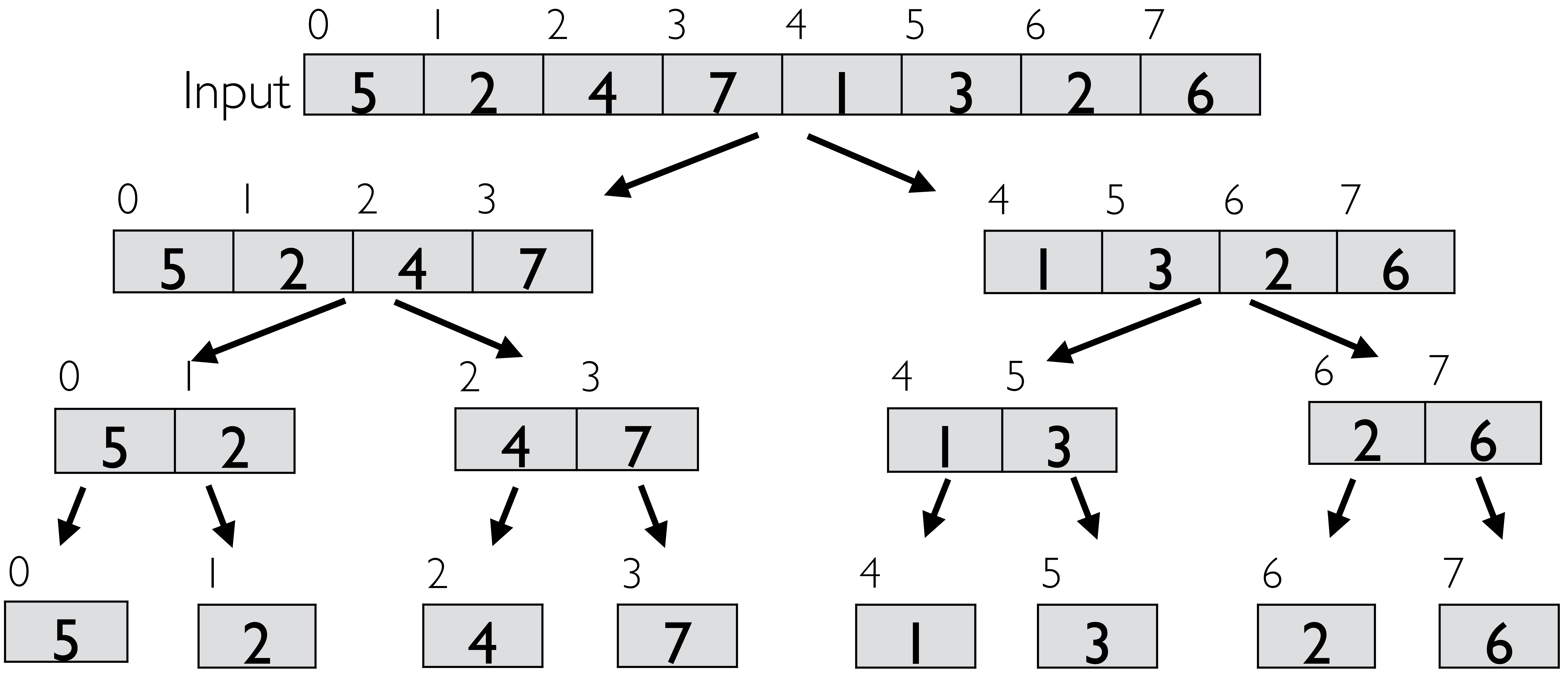
```
MergeSort(x):

if length of x is 1
    return x
else
    i = midpoint of x
    L = MergeSort(x[:i])
    R = MergeSort(x[i:])
    return Merge(L, R)
```

Recursively solve sub-problems

Complexity of merge sort

$N = 8 \rightarrow \log_2(8) = 3 \rightarrow$ divide input 3 times

Depth complexity is O(log n)



N elements

N/2 elements

N/4 elements

N/8 elements

At each level, process O(n) elements

Therefore, merge sort is O(n × log n)

OBJECT-ORIENTED PROGRAMMING (OOP)

What is OOP?

- **Object-oriented programming** (OOP) is a way of *organizing* data and code
- Built around the programming concepts of:
  - **Class** — A definition for a *type* of object
  - **Instance** — A *specific case* of that type of object
  - **Method** — Specialized *functions* for the object

Why use OOP?

- We've already been using OOP!
- Examples of built-in **classes** in Python:
  - Python 2+: `list, tuple, dict, set`
  - Python 3+: `int, float, str`
- In Python 3, *all* data types are classes

Built-in classes in Python

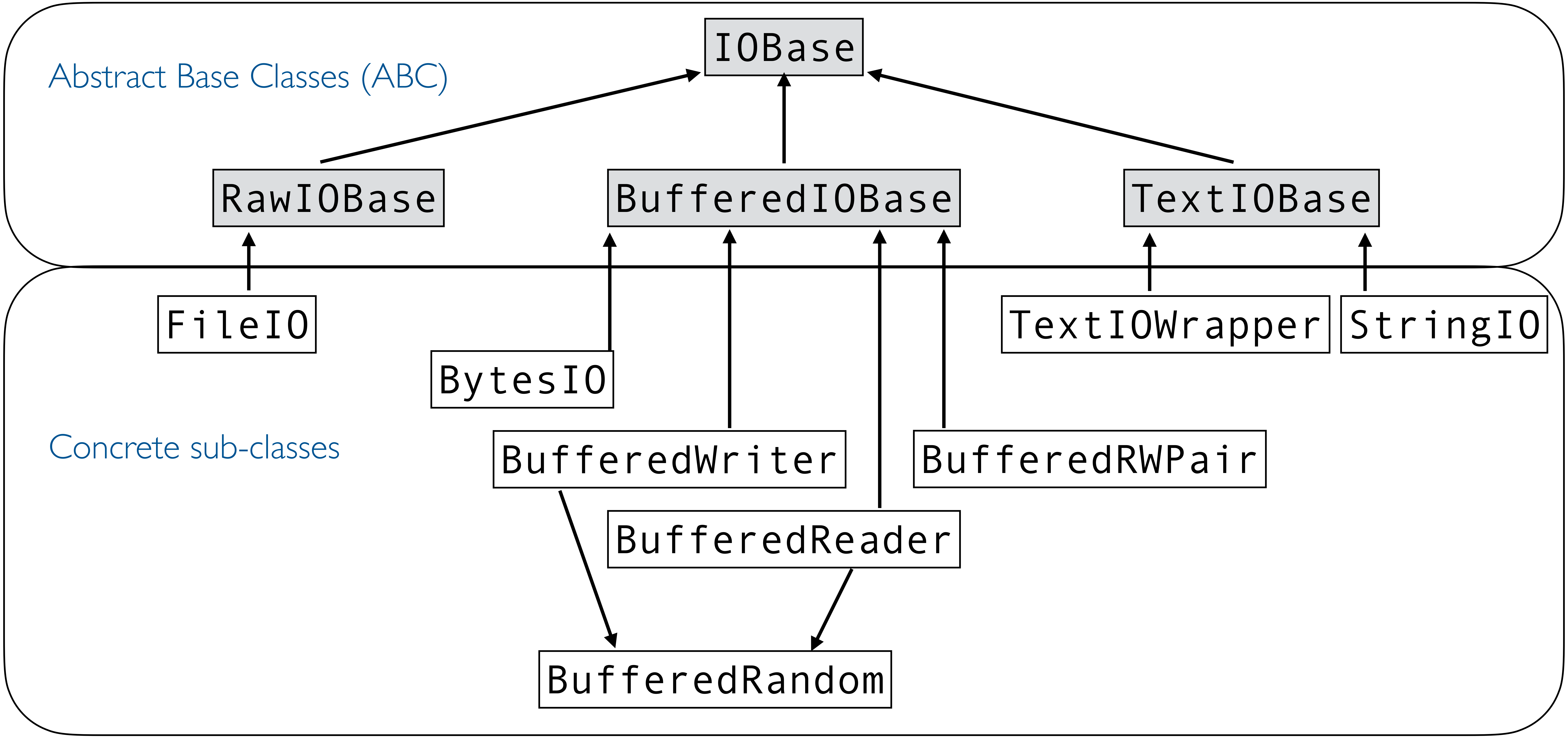
- `list` is a built-in *class* in Python
- `x` is an *instance* of a `list`
- `list.append()` is a *method*

```
list()

x = [1.11, 2.22, 3.33]

x.append(4.0)
```

Features of OOP

- Encapsulation
  - Bundle data and methods while hiding implementation details
- Composition
  - Objects may *contain* other objects to make complex objects
- Inheritance
  - Child classes may *inherit* behavior from their parent classes
- Polymorphism
  - Many data types can share a common interface

Hierarchy of Python I/O stream classes



Encapsulation

- Bundle together an object's *data* and the *methods* that operate on that data
- Hide implementation details from user
- Example: `TextIOWrapper`
  - I/O stream returned by `open()` on a text file
  - Contains buffer *data* and I/O *methods* (e.g., `read()`)
  - *Don't need to know how it's implemented*

Composition

- A complex object may be *composed* of multiple simpler objects
- Container "has-a" component relationship
- Example: `TextIOWrapper`
  - Provides a text interface to an I/O buffer
  - *Contains* a "wrapped" `BufferedIOBase` object
  - ...which in turn may *contain* a `RawIOBase` object

Inheritance

- Sub-classes (children) *inherit* methods from their super-classes (parents)
- Re-use implementation from super-class
- Use sub-classes to *specialize* behavior
- Example: `BufferedRandom`
  - Provides random access to an I/O buffer
  - "Is-a" `BufferedReader` *and* `BufferedWriter`
  - ... which *specialize* `BufferedIOBase`

Polymorphism

- Many types of objects may share a *common interface* for ease of use
- Caller doesn't need to care about or know the exact type of object
- Example: `IOBase`
  - Super-class for all I/O stream classes
  - All sub-classes implement `readlines()`, etc.
  - *No need to know exact class* to call `readlines()`

Class characteristics

- Use `class` keyword to define a *class*
  - Use `def` inside a `class` block to define a *method*
  - Special "magic" methods like `__init__` are *hooks*
  - Access instance *attributes* as `obj.attribute`
- *Good* classes follow OOP principles
  - Encapsulation & abstraction
  - Composition & inheritance
  - Polymorphism

Defining a class in Python

```
class Vector:

    def __init__(self, data):
        self.data = data

    def __str__(self):
        s = ",".join([str(x) for x in self.data])
        return "Vector<" + s + ">"

    def inner(self, y):
        prod = 0
        for xi, yi, in zip(self, y):
            prod += xi * yi
        return prod
```

Defining a class in Python

Class keyword

Name

Initialization method

String (print) method

Method defs

Inner product method

```
class Vector:
    def __init__(self, data):
        self.data = data

    def __str__(self):
        s = ",".join([str(x) for x in self.data])
        return "Vector<" + s + ">"

    def inner(self, y):
        prod = 0
        for xi, yi, in zip(self, y):
            prod += xi * yi
        return prod
```

Defining a class in Python (2)

Class keyword

Name

```
class Vector:
```

Refers to instance

"Hook" methods

```
    def __init__(self, data):
        self.data = data

    def __str__(self):
        s = ",".join([str(x) for x in self.data])
        return "Vector<" + s + ">"
```

Access attributes using dot notation

Regular method

```
    def inner(self, y):
        prod = 0
        for xi, yi, in zip(self, y):
            prod += xi * yi
        return prod
```

Defining methods in Python

- Function `def` inside `class` block creates a method
- First argument to a method should be `self`
  - Use `self` as a handle to the specific instance of the class
  - In practice, `foo(self, arg)` is called as `obj.foo(arg)`
- Special "magic" methods are hooks into Python
  - `__init__` is used to initialize instances of the class
  - `__add__` and `__mul__` implement `+` and `*`, etc.

Methods and `self`

- First argument to methods should be `self`
- Python passes the object as the first argument
  - Similar to `this` keyword in other languages like Java or C++
  - Passing of instance is *explicit* rather than *implicit* in Python
  - Use of "self" name is a (strong) convention, not a keyword
- Use as a handle to the "current" instance

Accessing attributes

- Access instance attributes using `obj.attribute`
  - Usually `self.attribute` inside a method
  - No private attributes — users can access them too!
- Get or set data attributes of an object
  - Typically, use `__init__` method to set initial values of attributes
  - Other methods may be used to change values of instance attributes
- Objects are *mutable* by default

"Magic" methods

- Nothing "magic" about double-underscore methods
  - "Dunder" methods are a core part of OOP system in Python
  - Use to make user-defined classes behave like built-in classes
- Special methods are hooks into Python operators
  - `__init__` is used to initialize instances of the class
  - `__add__` and `__mul__` implement `+` and `*`, etc.
- Only *really* need to know `__init__` for basic use

Special methods

Method	Implements
`__init__`	Object initialization
`__del__`	`del`
`__str__`	`print(), str()`
`__len__`	`len()`
`__iter__`	`iter()`
`__next__`	`next()`
`__reversed__`	`reversed()`
`__contains__`	`value in self`

Method	Implements
`__add__`	`self + value`
`__sub__`	`self - value`
`__mul__`	`self * value`
`__eq__`	`self == value`
`__lt__`	`self > value`
`__and__`	`self and value`
`__or__`	`self or value`
`__getitem__`	`self[i]`

. . . and many more!

Inheritance in Python

- Classes can *inherit* from super-classes
  - Enclose names of super-classes in parentheses after class name
  - E.g., `class SubName(SuperName)`
- Methods are inherited from the super-classes
  - Overwrite methods by re-defining them in sub-class
  - Use `super()` to access a *proxy instance* of the super-class to use the super-class versions of re-defined methods
- Possible to inherit from multiple classes

Defining a sub-class in Python

```
class Person:

    def __init__(self, name = "Jane Smith", uid = "000000"):
        self.name = name
        self.uid = uid

    def __str__(self):
        cls = self.__class__.__name__
        return "{}(name: {}, uid = {})".format(cls, self.name, self.uid)

class Employee(Person):

    def __init__(self, title = None, salary = 0, **kwargs):
        super().__init__(**kwargs)
        self.title = title
        self.salary = salary
```

Defining a sub-class in Python

Super-class of `Employee`

Default params

Default params

```
class Person:

    def __init__(self, name = "Jane Smith", uid = "000000"):
        self.name = name
        self.uid = uid

    def __str__(self):
        cls = self.__class__.__name__
        return "{}(name: {}, uid = {})".format(cls, self.name, self.uid)
```

Sub-class of `Person`

Packed `dict` of keyword args

```
class Employee(Person):

    def __init__(self, title = None, salary = 0, **kwargs):
        super().__init__(**kwargs)
        self.title = title
        self.salary = salary
```

Pass unpacked `dict` of keyword args

Super-class proxy

Review: Power of OOP

- Encapsulation
  - Bundle data and methods while hiding implementation details
- Composition
  - Objects may *contain* other objects to make complex objects
- Inheritance
  - Child classes may *inherit* behavior from their parent classes
- Polymorphism
  - Many data types can share a common interface