Algorithms and complexity

Kylie A. Bemis

Northeastern University
Khoury College of Computer Sciences

Goals for today

- Review of functions
- Intro to algorithms
- Time and complexity

REVIEW:
FUNCTIONS

Why functions?

- Code should be **reusable**!
- *Decomposition* creates structure
    - **Self-contained** chunk of code
    - **Coherent** and **organized** design
- Performs a **single task** using *input*
- Returns a value as *output*

Using functions

- We use many functions (e.g., `print()`)
- *Abstraction* supports usability
  - Functions are a "**black box**" for users
  - No need to know implementation details
- Supported usage should be **documented**
  - Function specification
  - Docstring

Function characteristics

- Functions in Python have:
  - Name
  - **Parameters** (0 or more)
  - **Docstring** (optional, but recommended)
  - Body (implementation)
  - **Return** value
- *Good* functions are intuitive to use

Defining a function in Python

```python
def mysum( x ):
    """
    Sums values of an iterable
    param x: An iterable to sum the values
    returns: The sum
    """
    xsum = 0
    for xi in x:
        xsum += xi
    return xsum
```

Defining a function in Python

Function keyword

Name

Parameter

```
def mysum( x ):
    """
    Sums values of an iterable
    param x: An iterable to sum the values
    returns: The sum
    """
    xsum = 0
    for xi in x:
        xsum += xi
    return xsum
```

Docstring

Body

Return value

```
mysum([1, 2, 3])
```

Usage (later in code)

Scope in functions

- **Scope** is how a program looks up names
- Variables may be found in 3 scopes:
    - *Local* variables are assigned inside a function `def`
    - *Nonlocal* variables are assigned in an *enclosing* `def`
    - *Global* variables are assigned in the top-level module
- Variables with the same name, but different scopes, are *different* variables

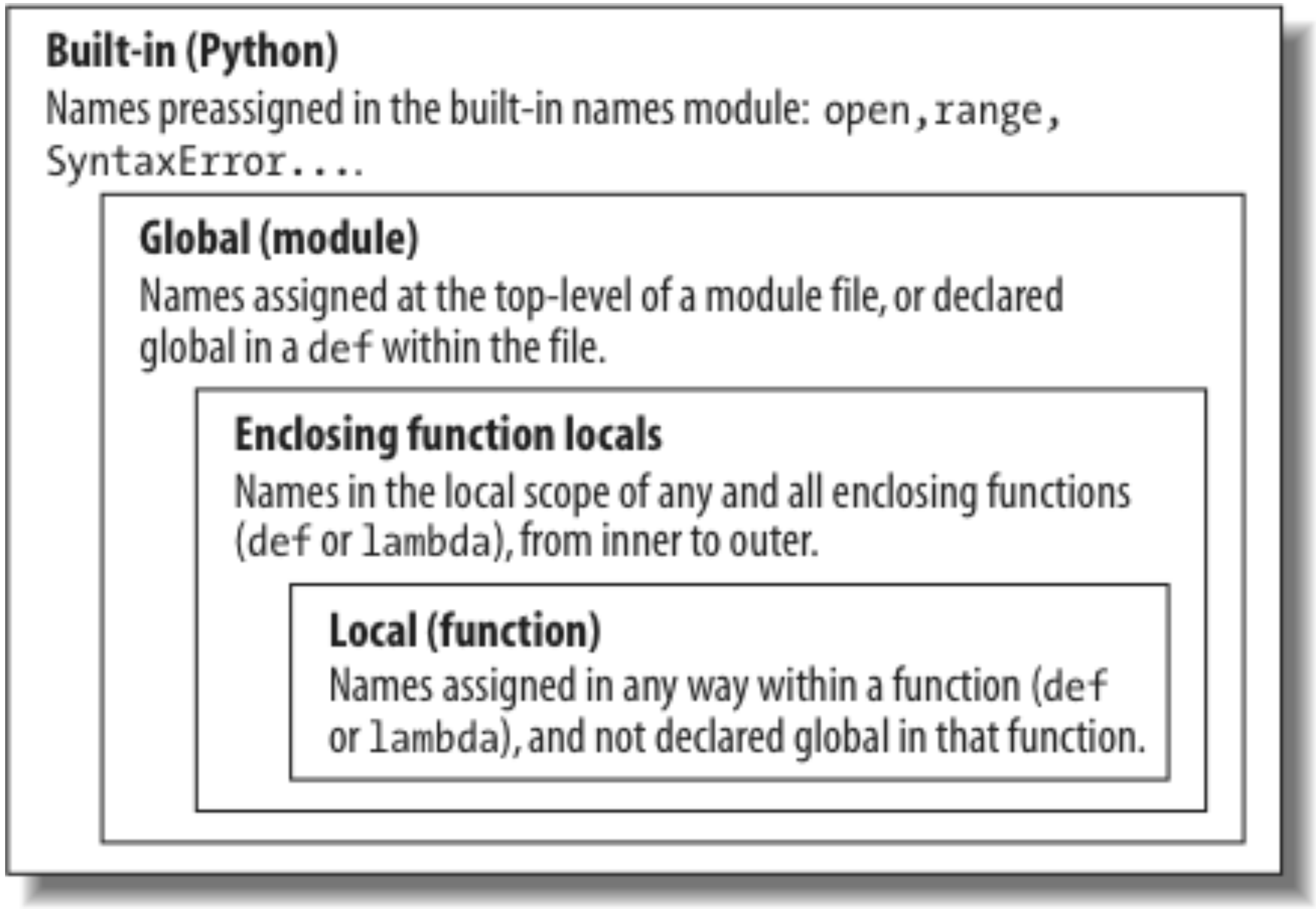
Scope example

```
# Global: X is assigned at top-level
X = 99

def func(Y):    # Local: Y and Z assigned in def
    Z = X + Y   # Accesses global X
    return Z

func(1) # returns 100
```

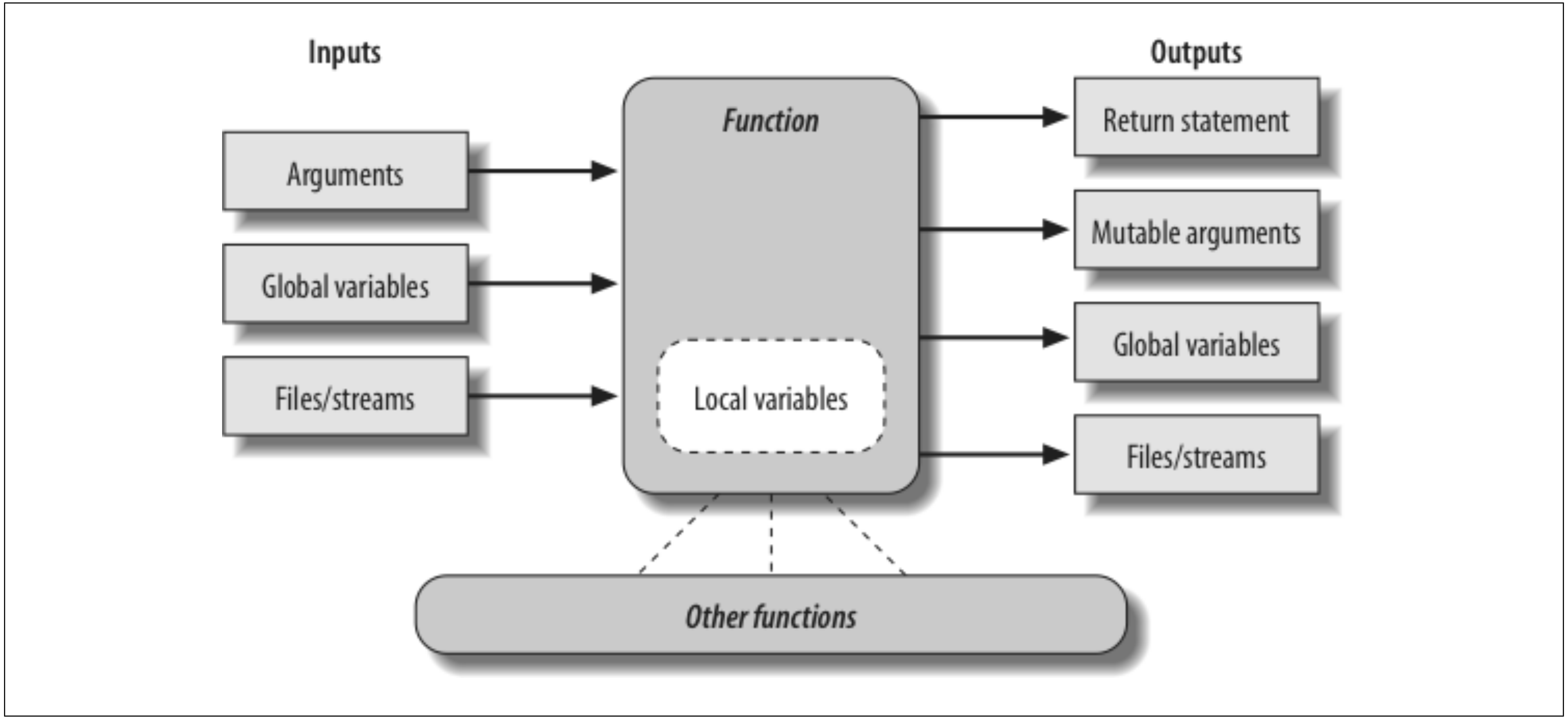
Another look at Python scopes



Design of good functions

- Organize your function's inputs and outputs
  - Arguments should be inputs
  - Use `return` for outputs
  - Only modify mutable arguments when *expected*
  - Avoid global variables for inputs or outputs
- Functions should have a clear, single purpose
- Function code should be relatively small

Functions in Python



Function recursion

- Functions are allowed to call *themselves*
- Good for solving problems of recurrent relationships
- Fibonacci:
  - $f(1) = 1, f(2) = 1$
  - $f(n) = n + f(n - 1)$
- Factorial:
  - $f(1) = 1$
  - $f(n) = n \times f(n - 1)$

Factorial example

```
def fact(n):
    """
    Factorial of n (i.e., n!)
    param n: A number
    returns: Factorial of n
    """
    if n == 1:
     return n
    else:
     return n * fact(n-1)
```

INTRO TO ALGORITHMS

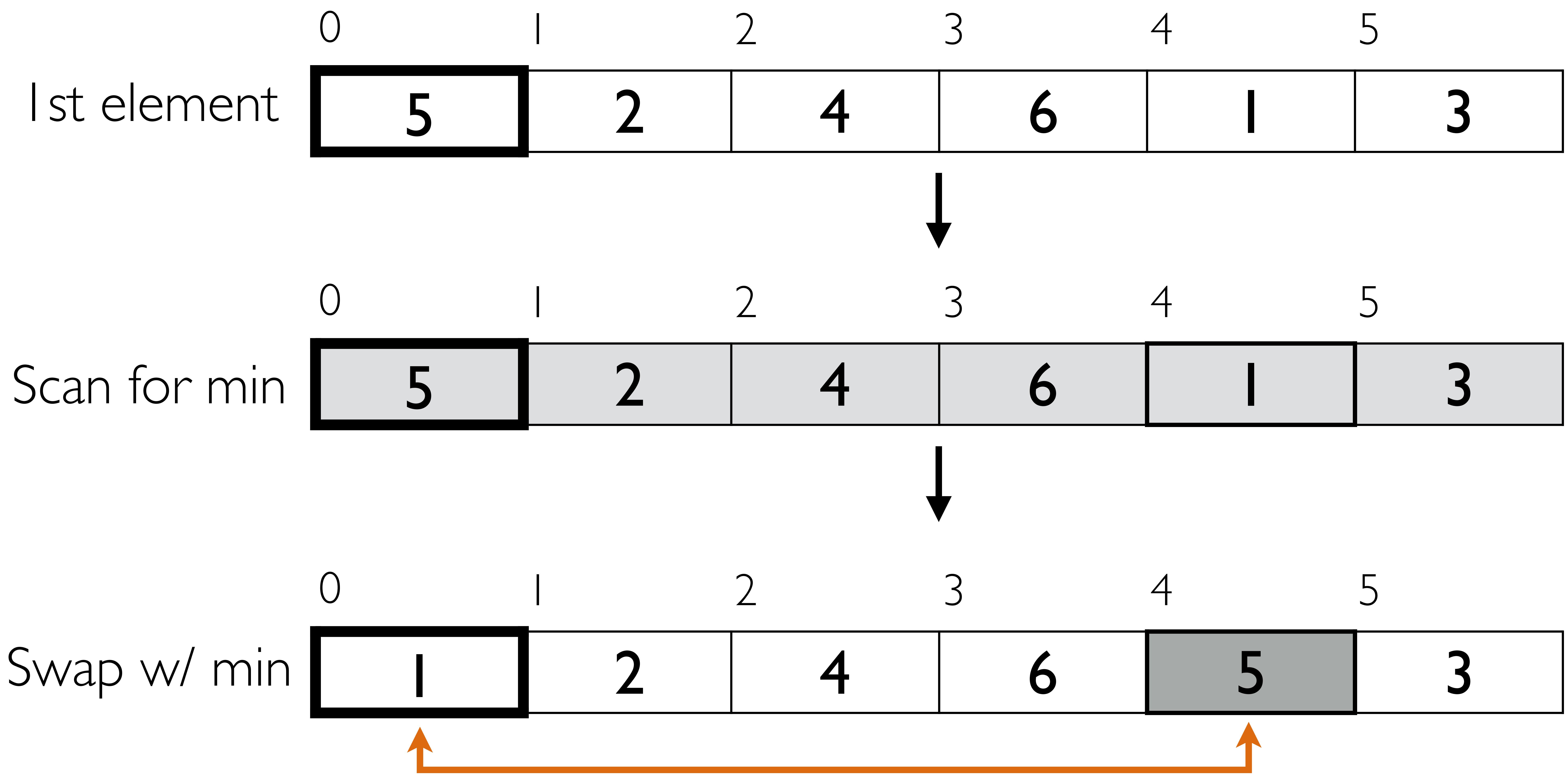
Sort an array in increasing order?

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

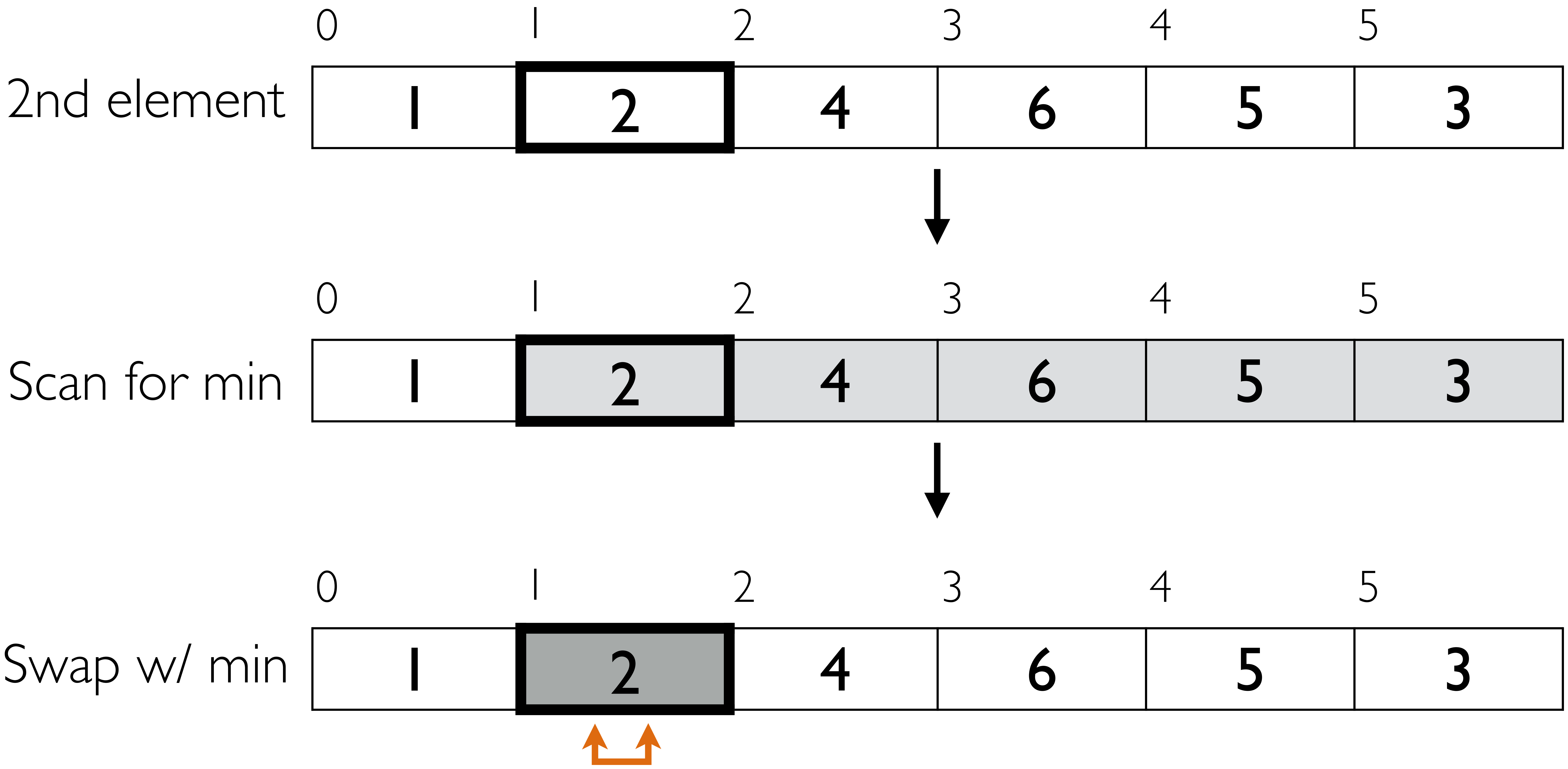
↓

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

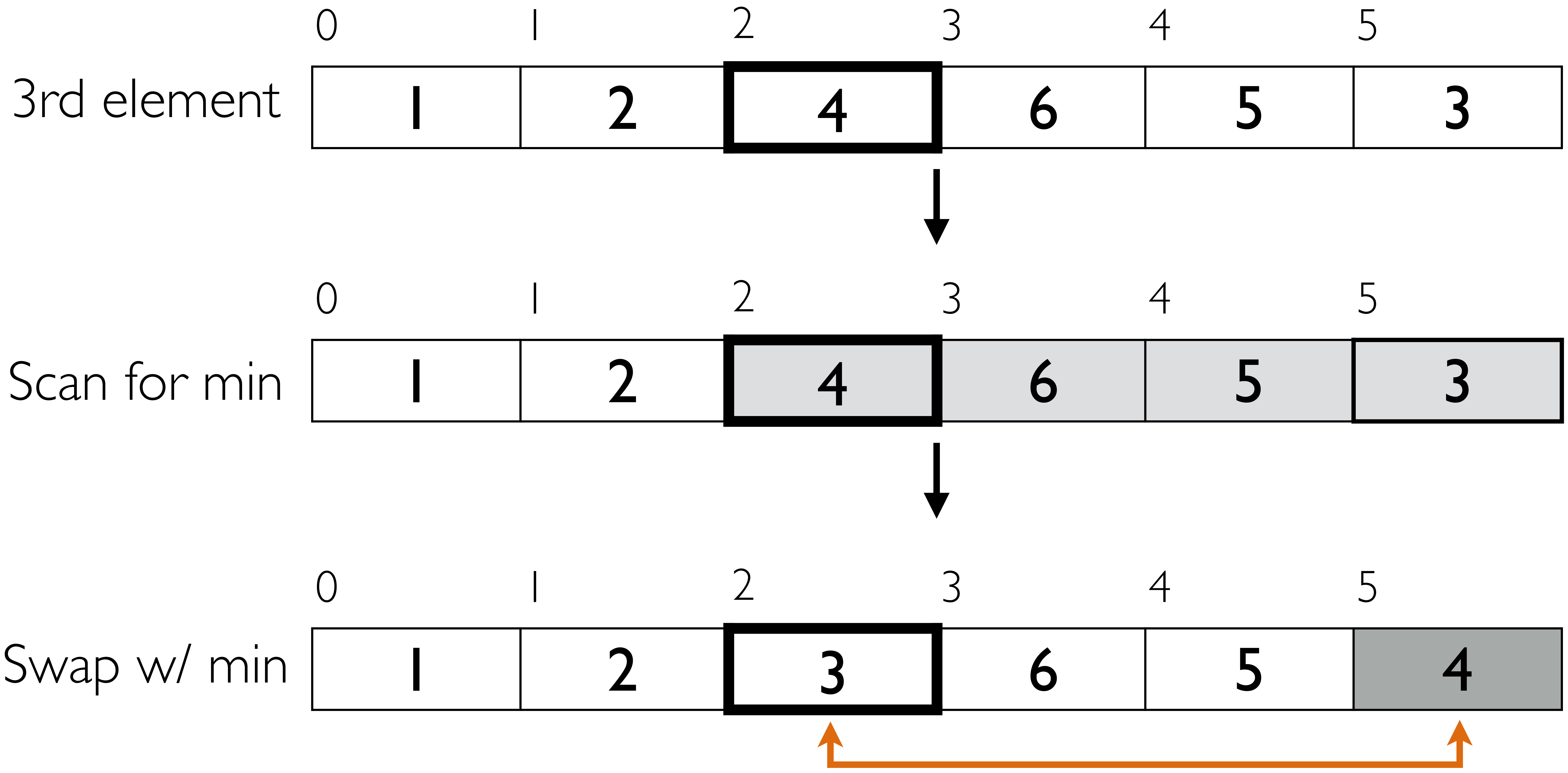
1. Select the smallest element



2. Select the second smallest element



3. Select the third element



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]
```

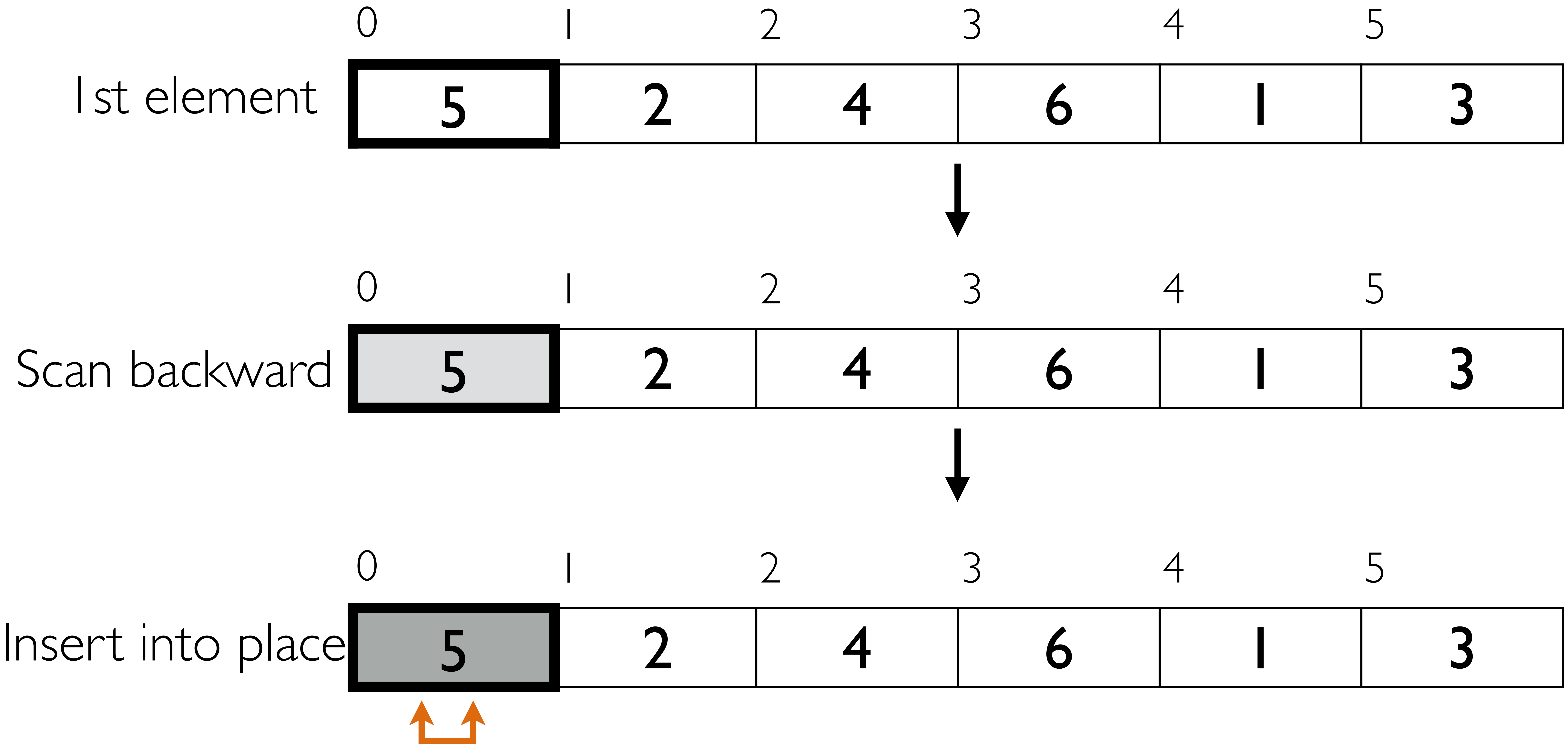
Selection sort

```
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)
```

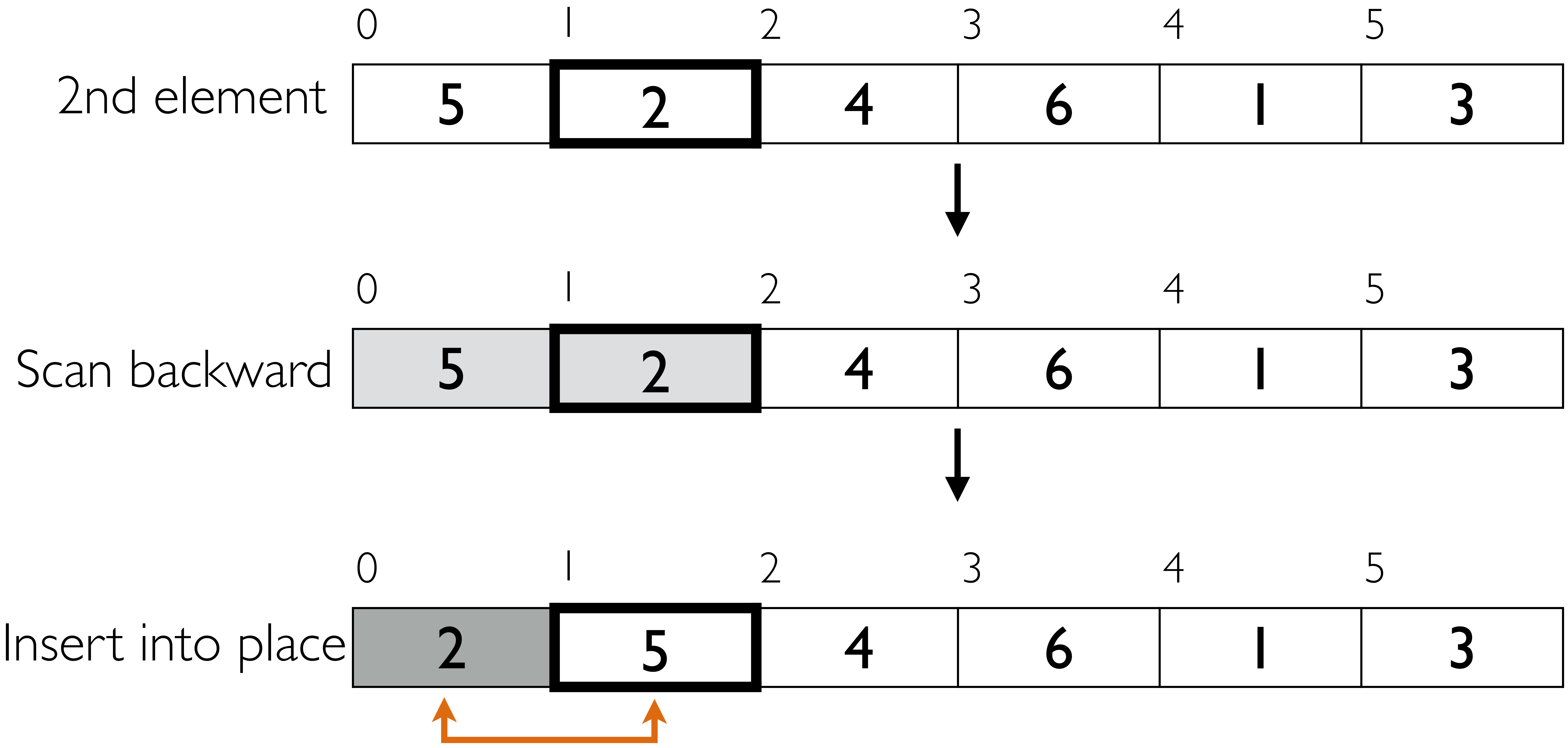
Improving selection sort

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

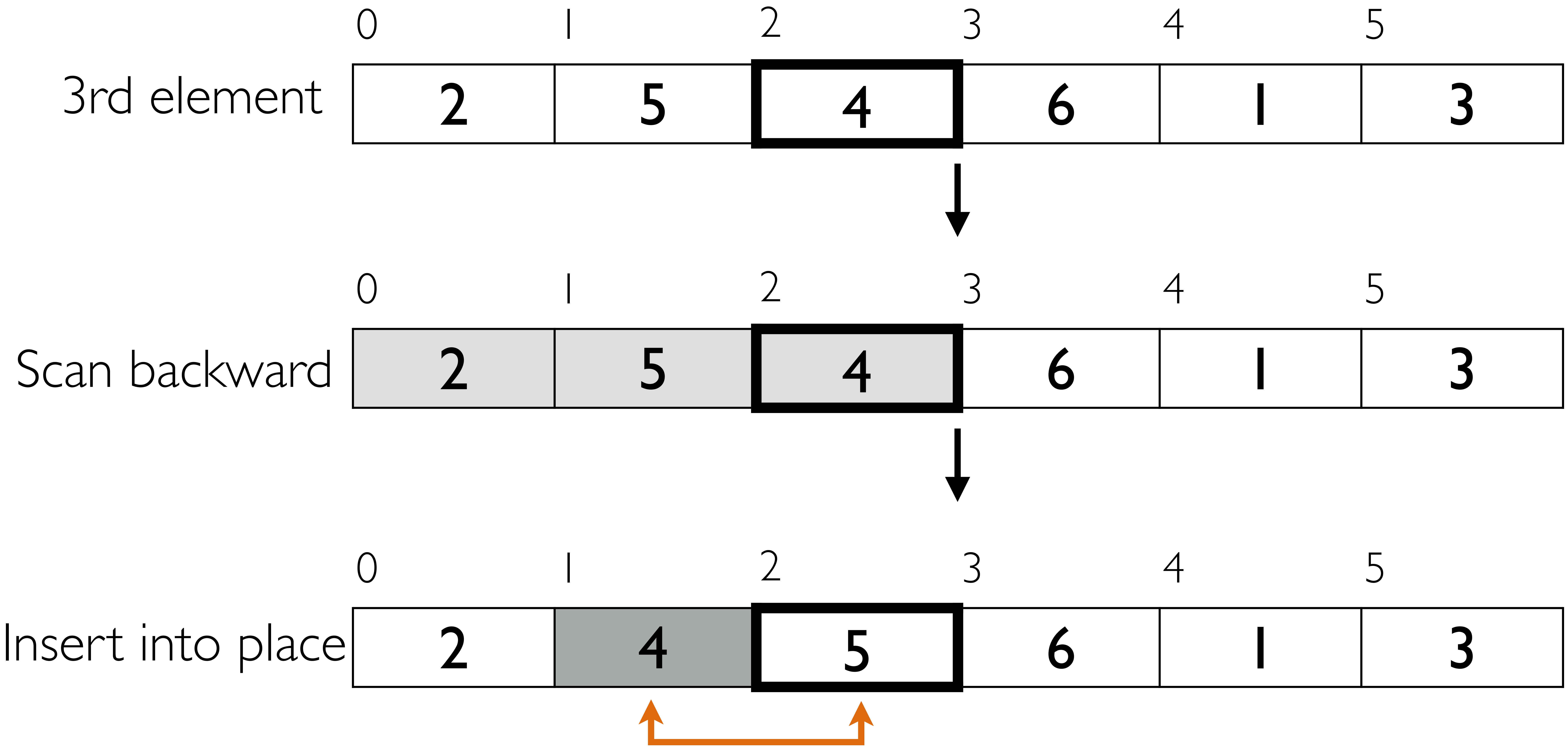
1. Insert first element into subarray



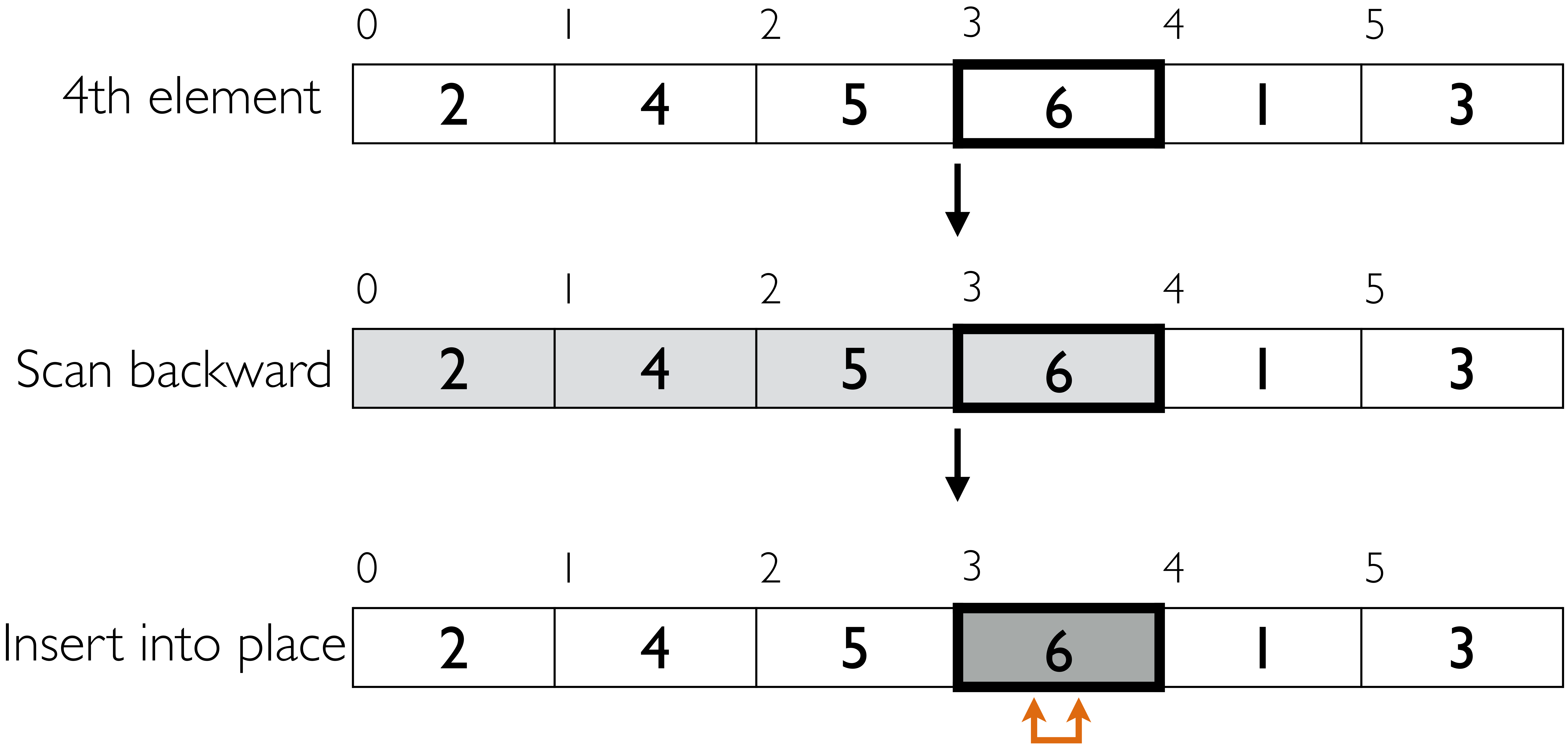
2. Insert second element into subarray



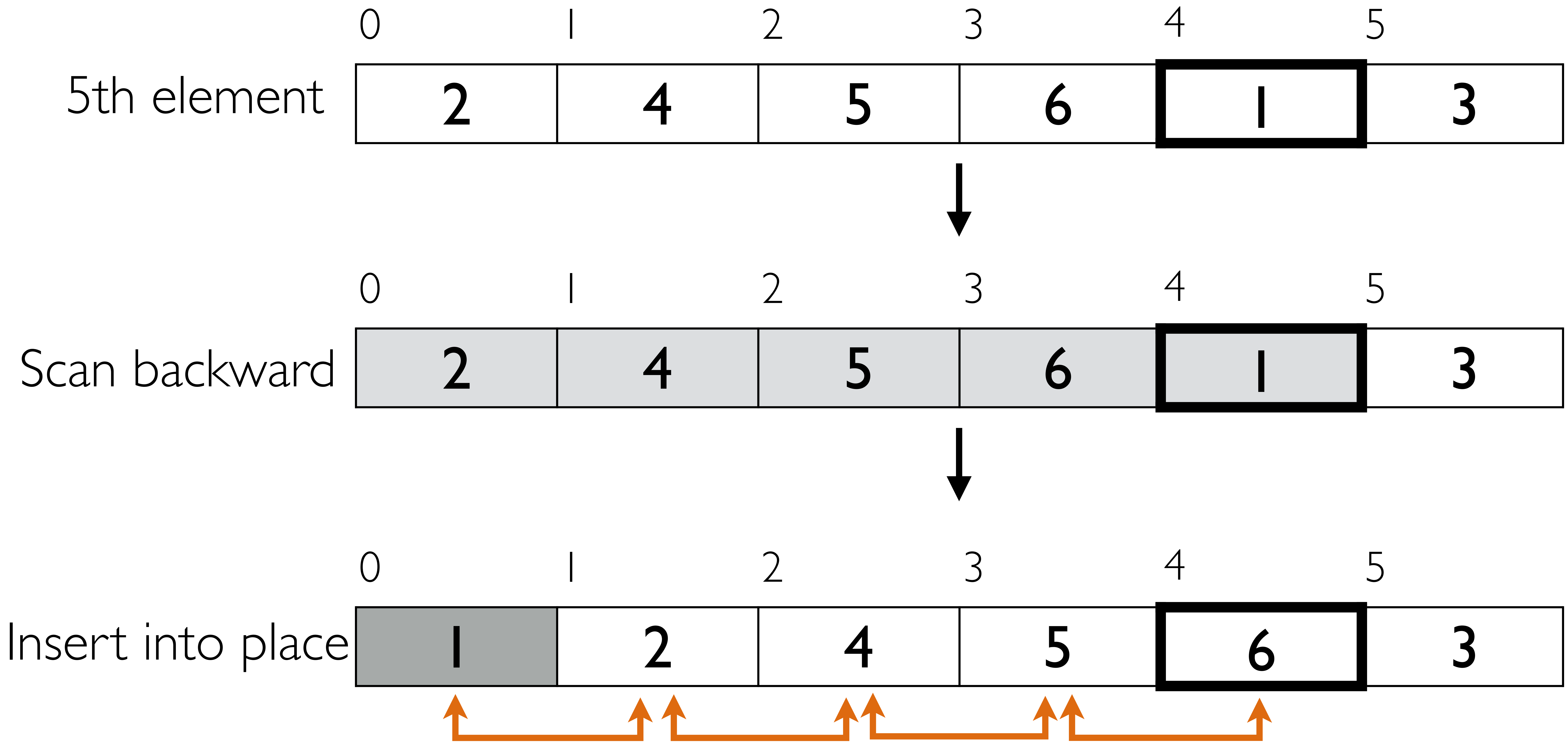
3. Insert third element into subarray



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
```

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 sorted sublist x[:i]
      while j >= 0 and x[i] < x[j]:
        if x[i] < x[j]:
          swap(x, i, j)
          i = j # note: won't affect next iteration's 'i'!
        j -= 1
```

Divide and conquer

- Can we divide the program into smaller sub-problems?
- Sub-problems should be easier to solve
- Re-combine the output of sub-problems into a complete solution

Sort array with divide and conquer?

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

	0	1	2	3	4	5	6	7
Output	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

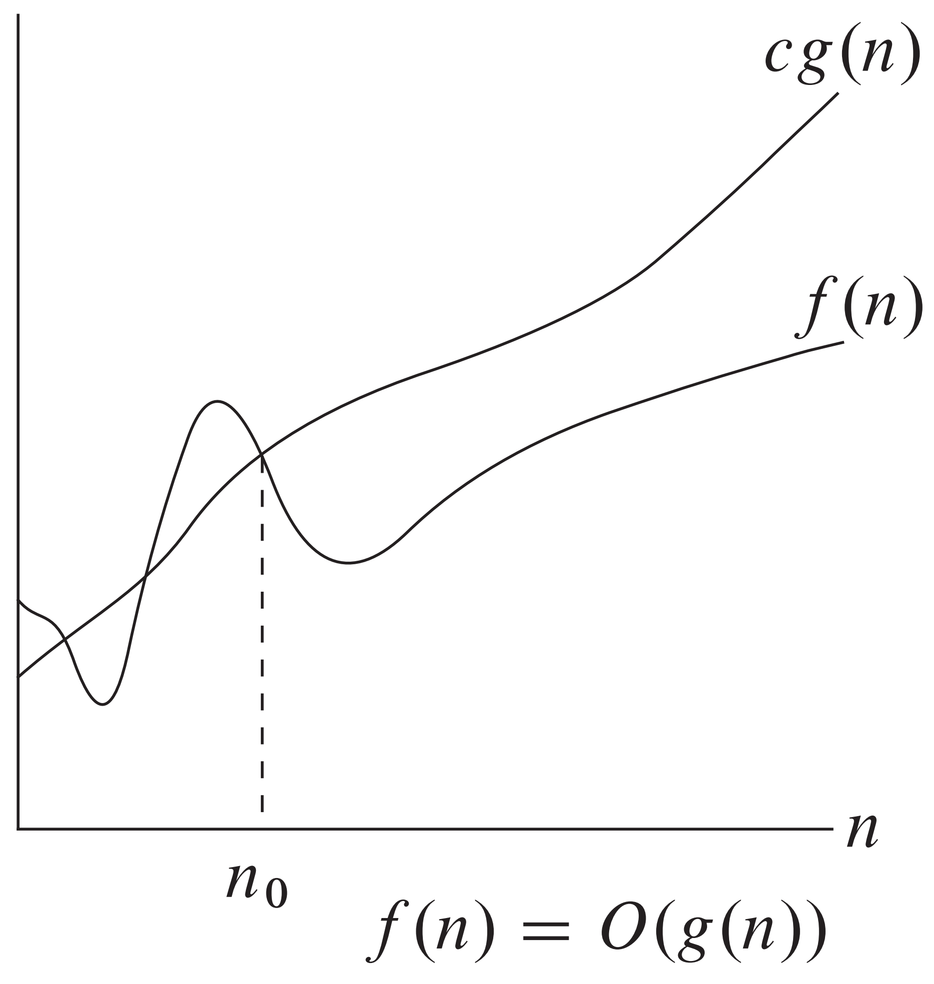
```
MergeSort(x):

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)
```

TIME AND COMPLEXITY

Measuring algorithmic complexity

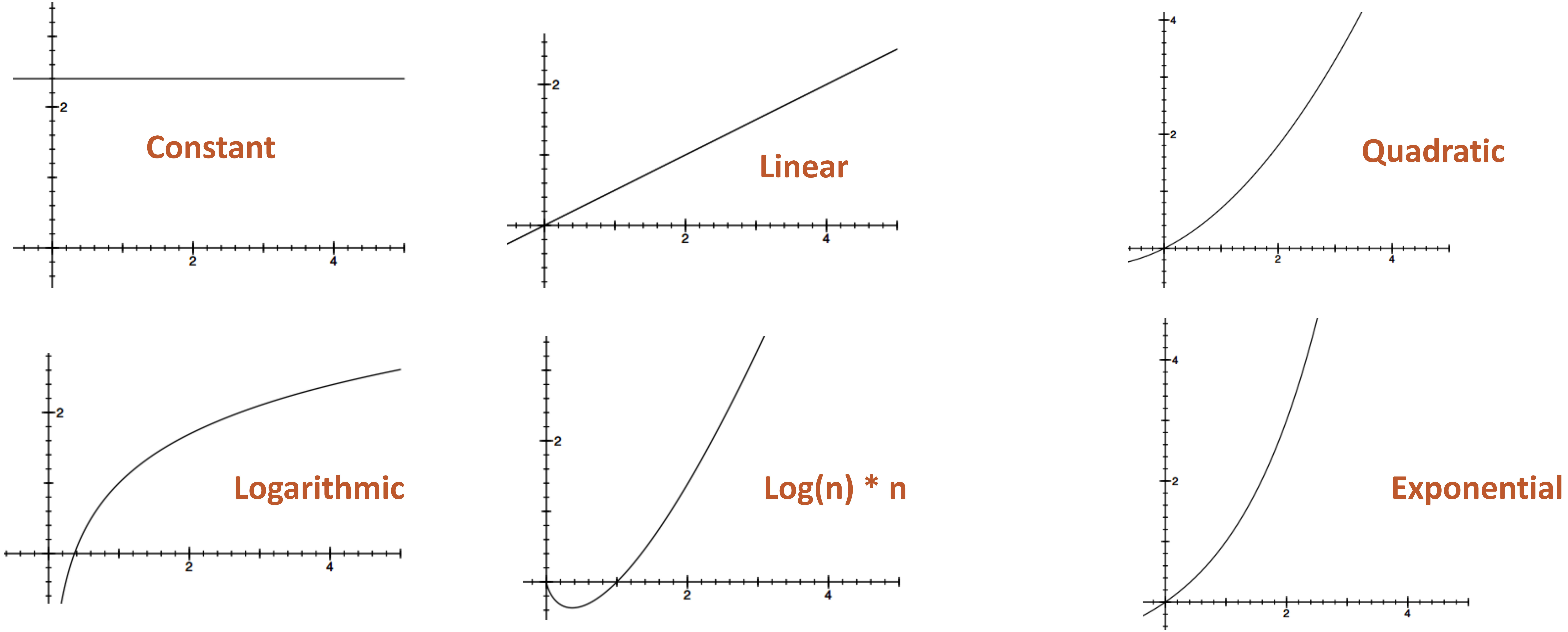
- *Upper bound* on amount of time for an algorithm to complete for input size ***n***
- Asymptotic *upper bound* described as $O(g(n))$
- A function $f(n)$ is $O(g(n))$ if
  - For some constant c and all values of $n \geq$ some value n_0
  - $f(n) \leq c \times g(n)$



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

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

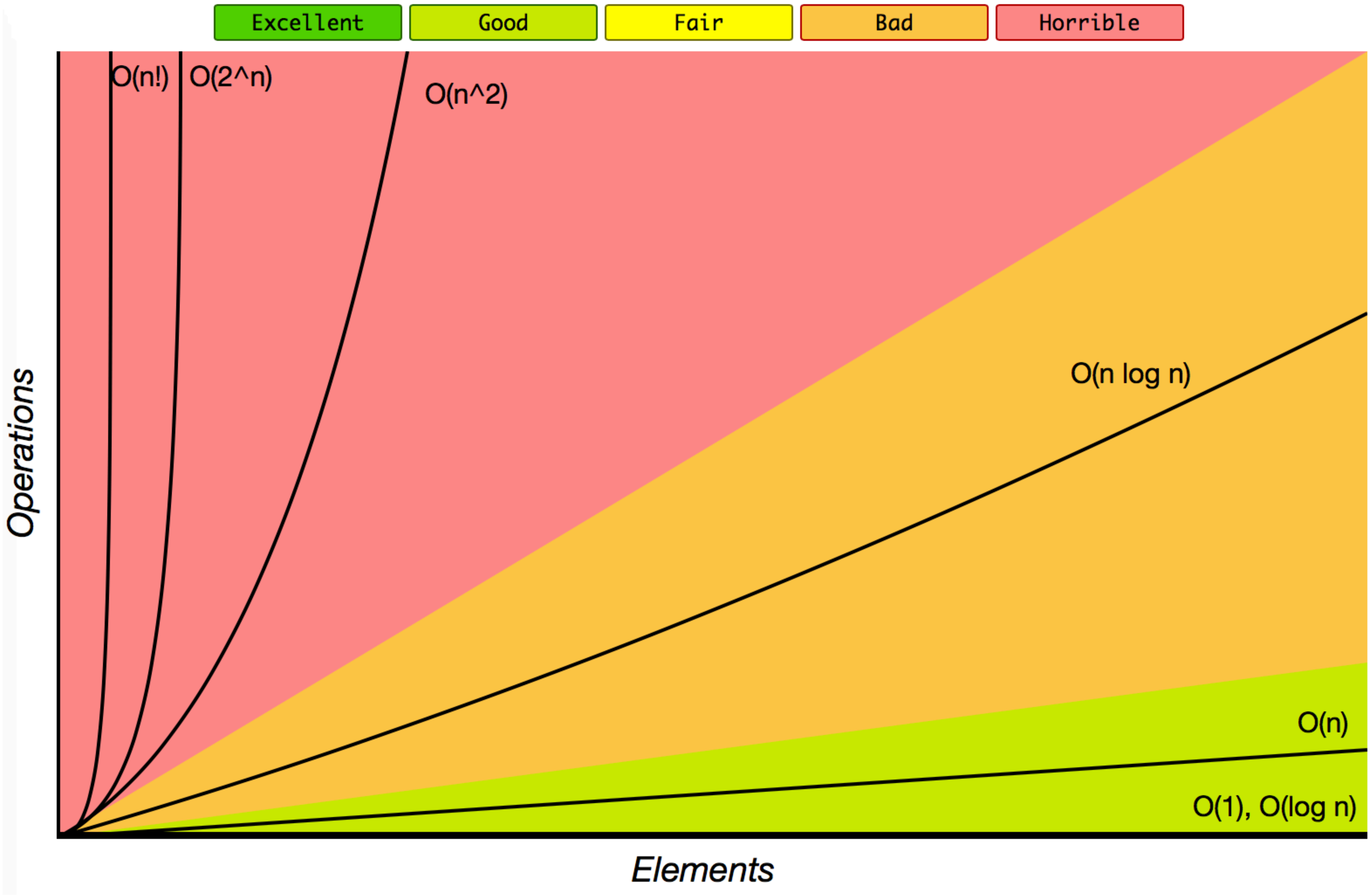
Growth rates of functions



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/

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)

↓

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

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 ...
      while j >= 0 and x[i] < x[j]:
         if x[i] < x[j]:
            swap(x, i, j)
            i = j # note: ...
         j -= 1
```

Outer loop grows as O(n)

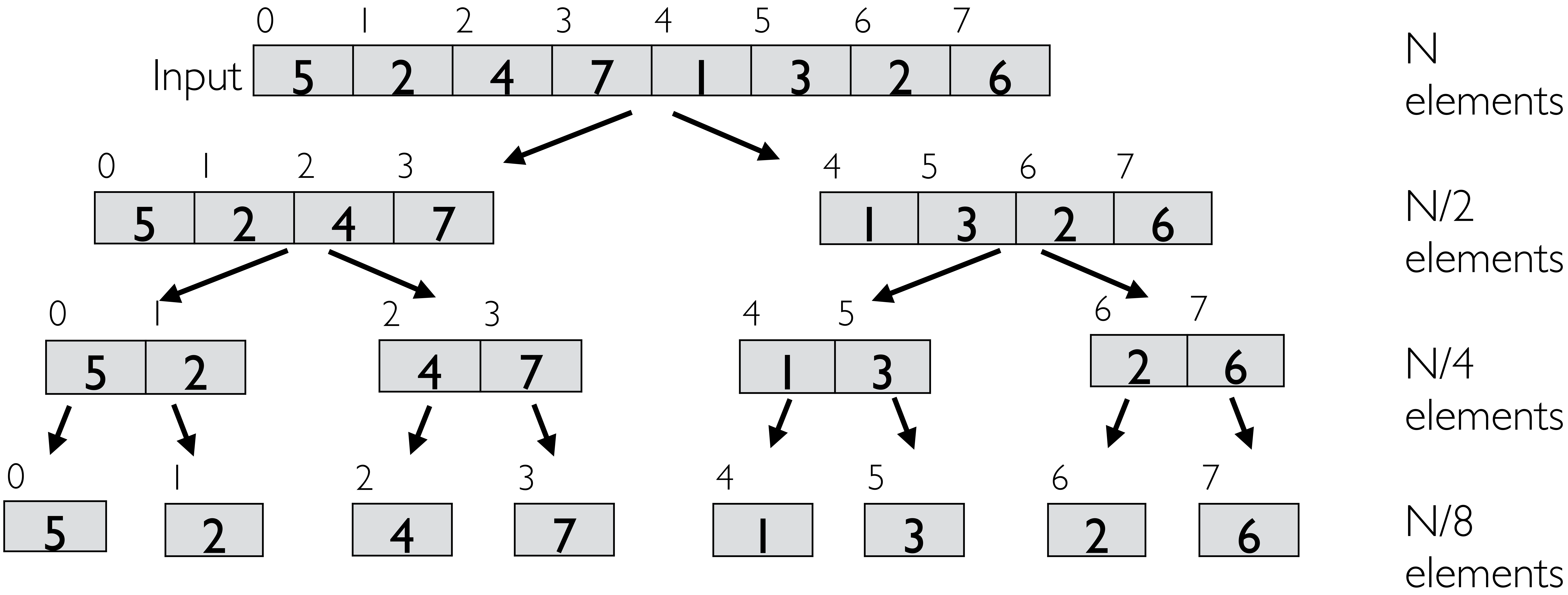
Inner loop grows as O(n)

Therefore, insertion sort is also O(n×n) = O(n^2)

Complexity of merge sort

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

Depth complexity is $O(\log n)$



At each level, process $O(n)$ elements

Therefore, merge sort is $O(n \times \log n)$

Common patterns of complexity

- Constant time
  - Operations that do not depend on input size
- Linear and polynomial time
  - Simple loops *dependent on input size* are linear: $O(n)$
  - *Nested* loops are polynomial: $O(n) \rightarrow O(n^2) \rightarrow O(n^3)$
- Logarithmic time
  - Algorithms that *repeatedly divide input*
  - Don't forget to consider sub-operations!