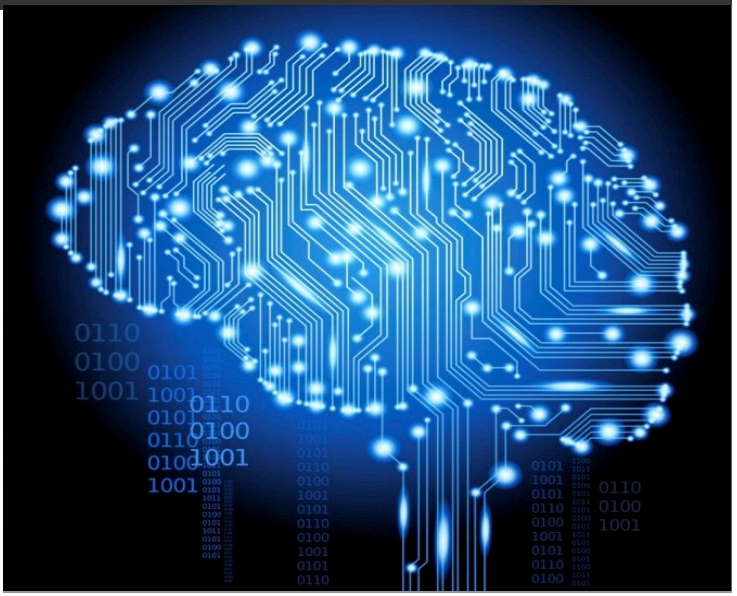
MONASH University

Information Technology

FIT1008/2085 Sorting

Prepared by:
Maria Garcia de la Banda
Revised by D. Albrecht, J. Garcia



Objectives for these lecture

- **To study in some depth three basic sorting algorithms:**
  - Bubble Sort
  - Selection Sort
  - Insertion Sort
- **To be able to implement, use and modify them in Python**
- **To use them to reason about some properties of algorithms:**
  - Invariants
- **To be able to use some of the invariants to improve them**
- **In the next lesson, we will use these algorithm to illustrate complexity**

Sorting lists (increasing order)

Example:

[6,4,2,1,3,5] ⟶ [1,2,3,4,5,6]

Sorting Lists (increasing order)

- **Input:**
  - A list (not necessarily sorted) of 'orderable' element types
  - For example, in Python:
    - `the_list = [5,1.5,3,-4.0]` is fine
    - `the_list = [1,'hj',0,'j']` is not
      - Unless you define your own comparison function
- **Output:**
  - A list with the same elements as the input list BUT sorted in increasing order
- **Sorted according to what?**
  - Right now, we will assume it is sorted by the element
  - In the future, things will get a bit more interesting

MONASH University

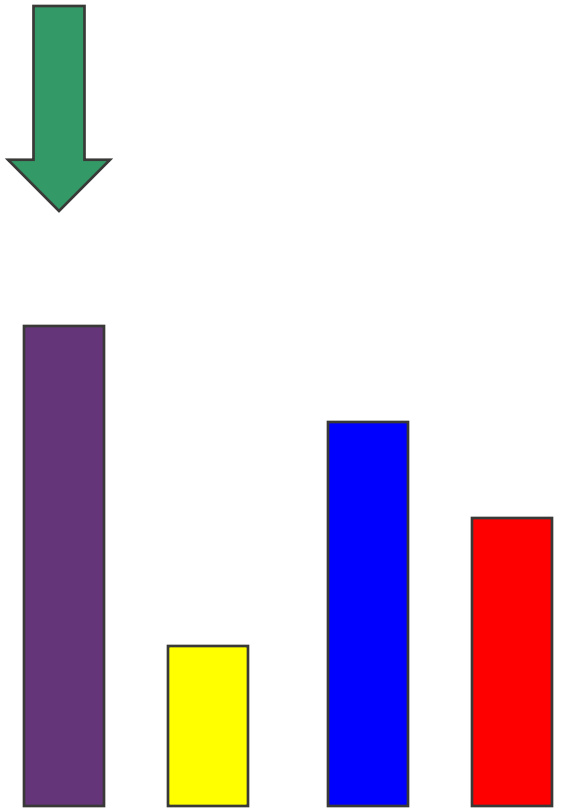
Bubble Sort

Bubble Sort

- **Very simple, so perfect for thinking about sorting**
- **We have used the naïve version in the interview prac**
- **Do the following in every iteration:**
  - Start at the leftmost element X
  - Compare X to the element Y to its right
  - If X > Y swap them, otherwise don't
  - Move one position to the right

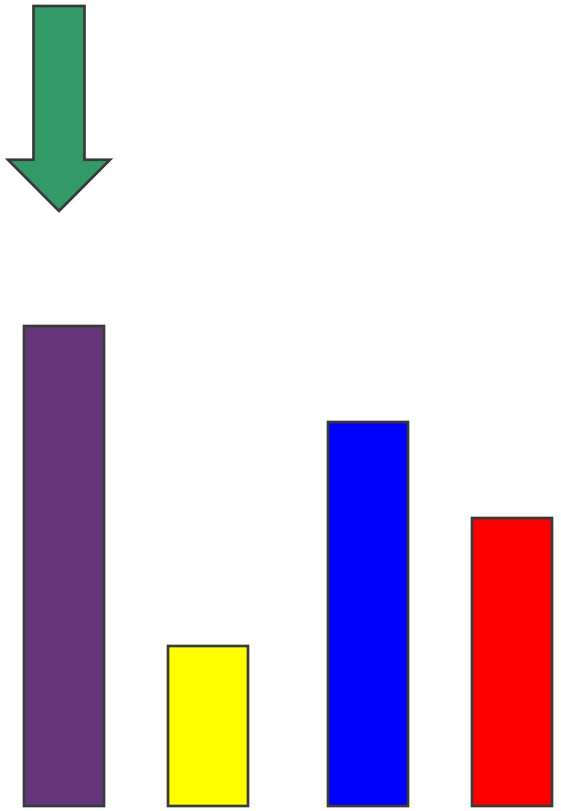
Bubble Sort Iteration: Example

- Start at leftmost X
- If X > Y, swap them
- Move to right



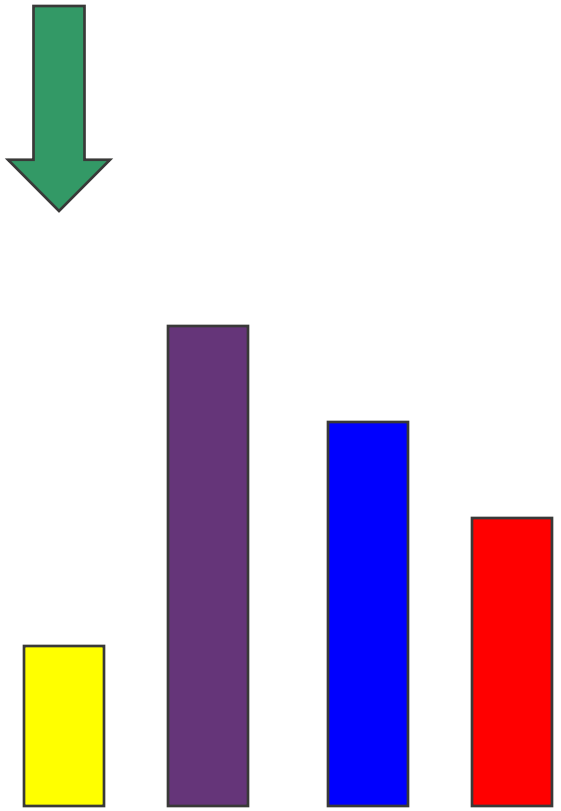
Bubble Sort Iteration: Example

- Start at leftmost X
- If X > Y, swap them
- Move to right



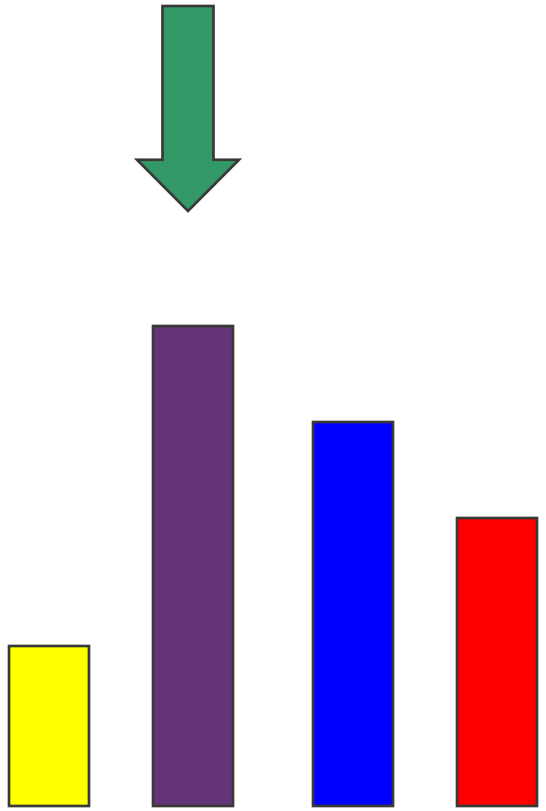
Bubble Sort Iteration: Example

- Start at leftmost X
- If X > Y, swap them
- Move to right



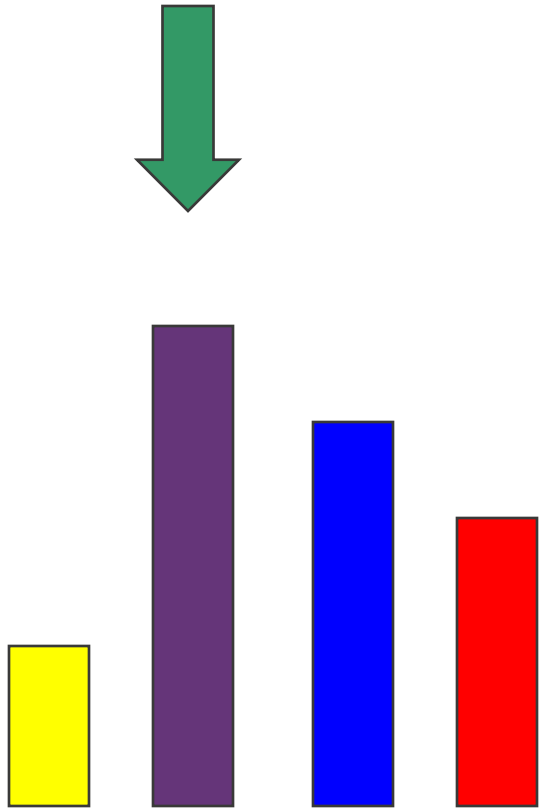
Bubble Sort Iteration: Example

- Start at leftmost X
- If X > Y, swap them
- Move to right



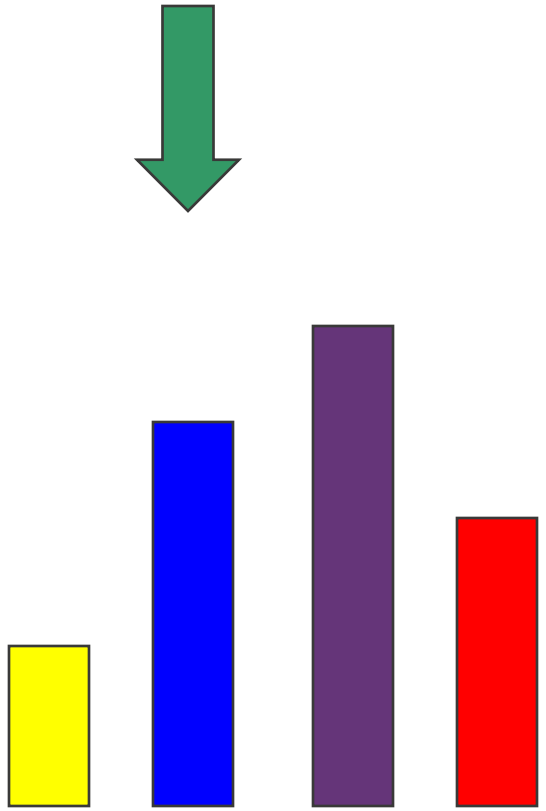
Bubble Sort Iteration: Example

- Start at leftmost X
- If X > Y, swap them
- Move to right



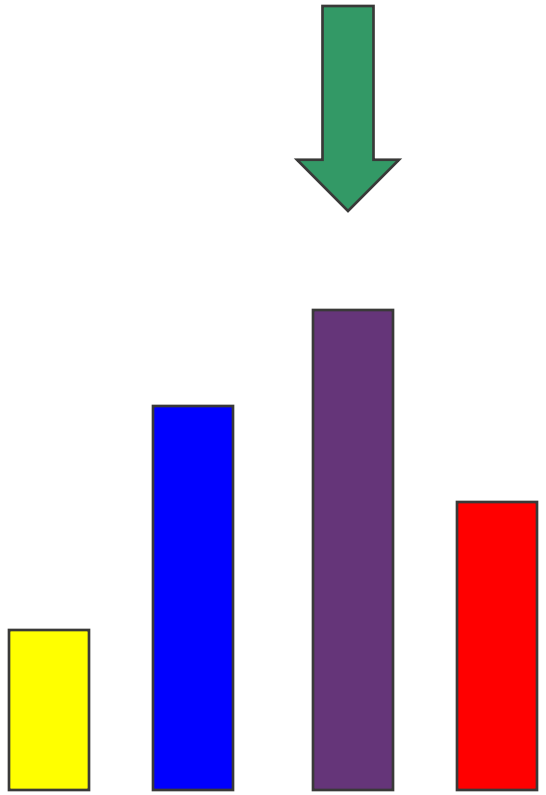
Bubble Sort Iteration: Example

- Start at leftmost X
- If X > Y, swap them
- Move to right



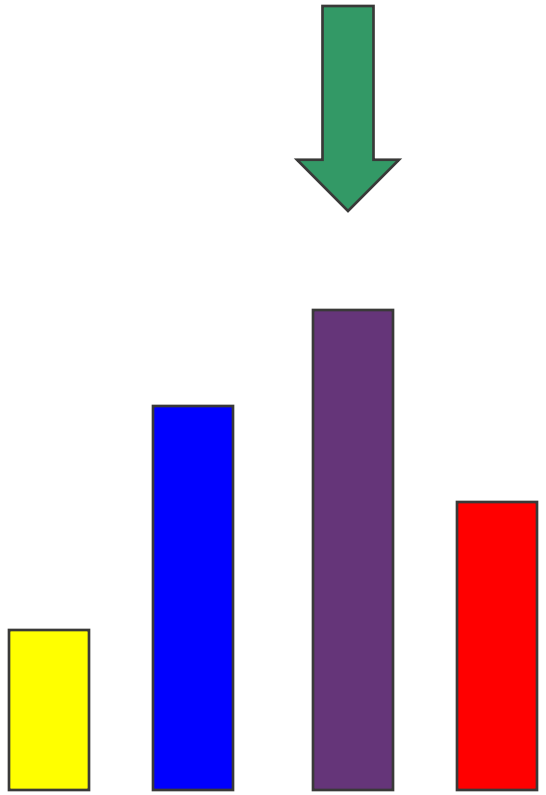
Bubble Sort Iteration: Example

- Start at leftmost X
- If X > Y, swap them
- Move to right



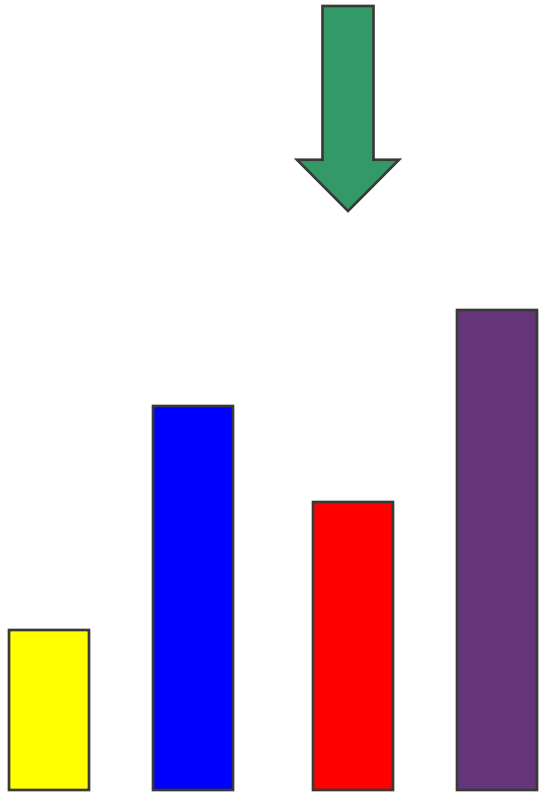
Bubble Sort Iteration: Example

- Start at leftmost X
- If X > Y, swap them
- Move to right



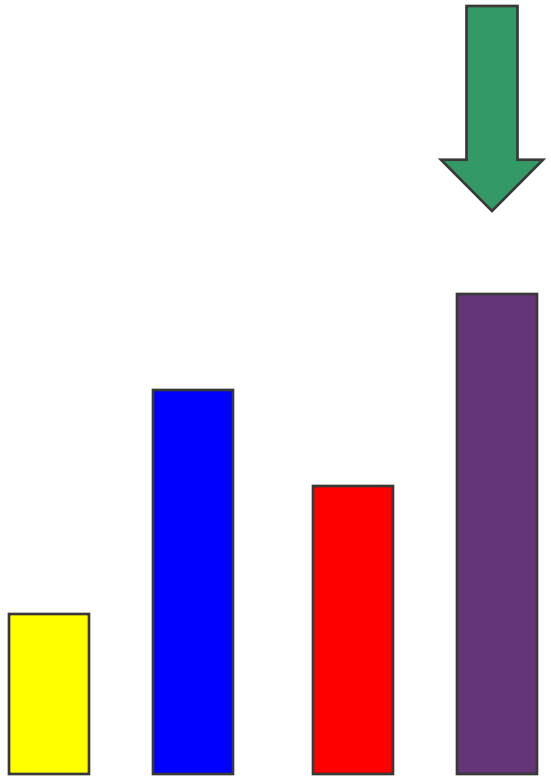
Bubble Sort Iteration: Example

- Start at leftmost X
- If X > Y, swap them
- Move to right



Bubble Sort Iteration: Example

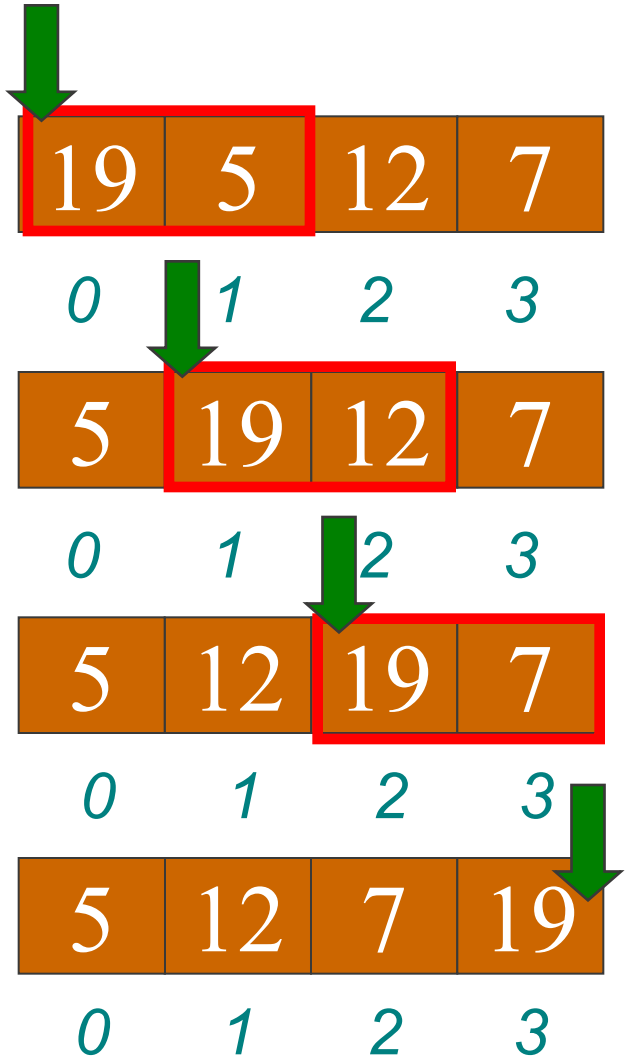
- Start at leftmost X
- If X > Y, swap them
- Move to right



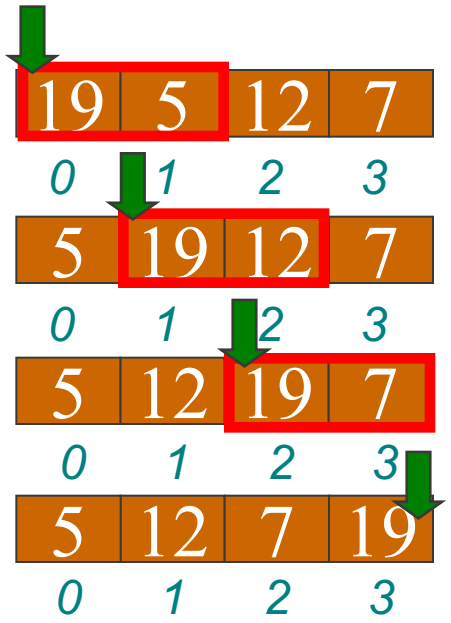
Bubble Sort: Invariants

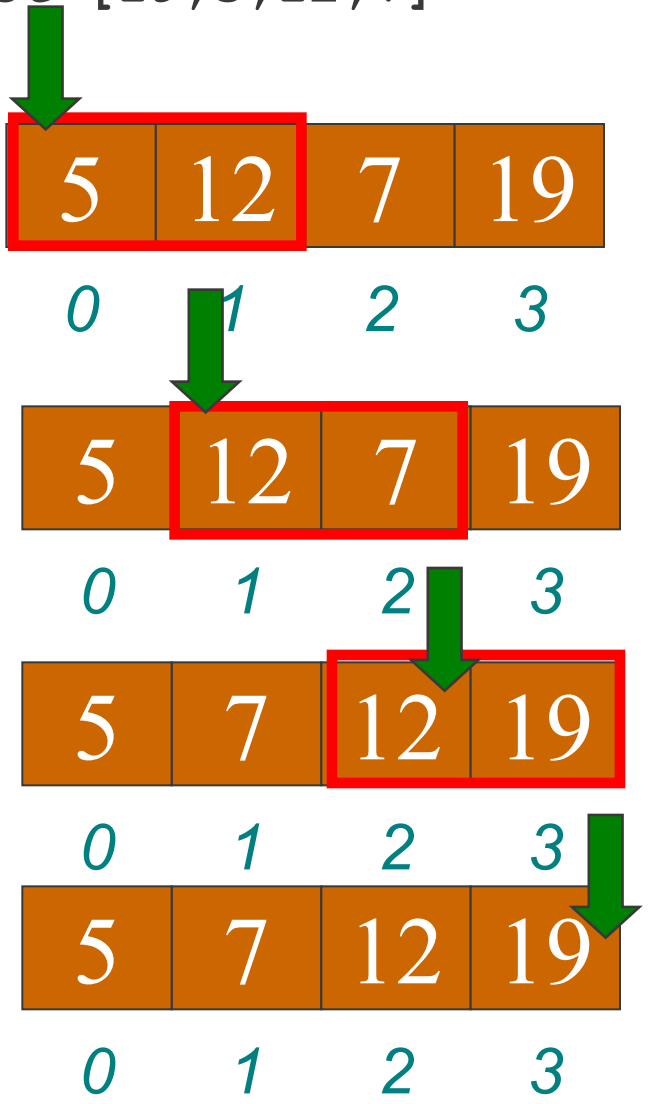
- **Invariant: property that remains unchanged**
  - At a particular program point, or throughout an algorithm, a function, a module …
- **In bubble sort there are many invariants:**
  - Example: after every traversal, the list has the same elements
  - Also: in each traversal at most n-1 swaps are performed, where n is the length of the list
- **One invariant is particularly interesting:**
  - After every traversal, the largest yet unsorted element gets to its final place
- **It tells us the maximum number of traversals needed to sort**
  - n-1

Bubble Sort – Example: `the_list=[19,5,12,7]`

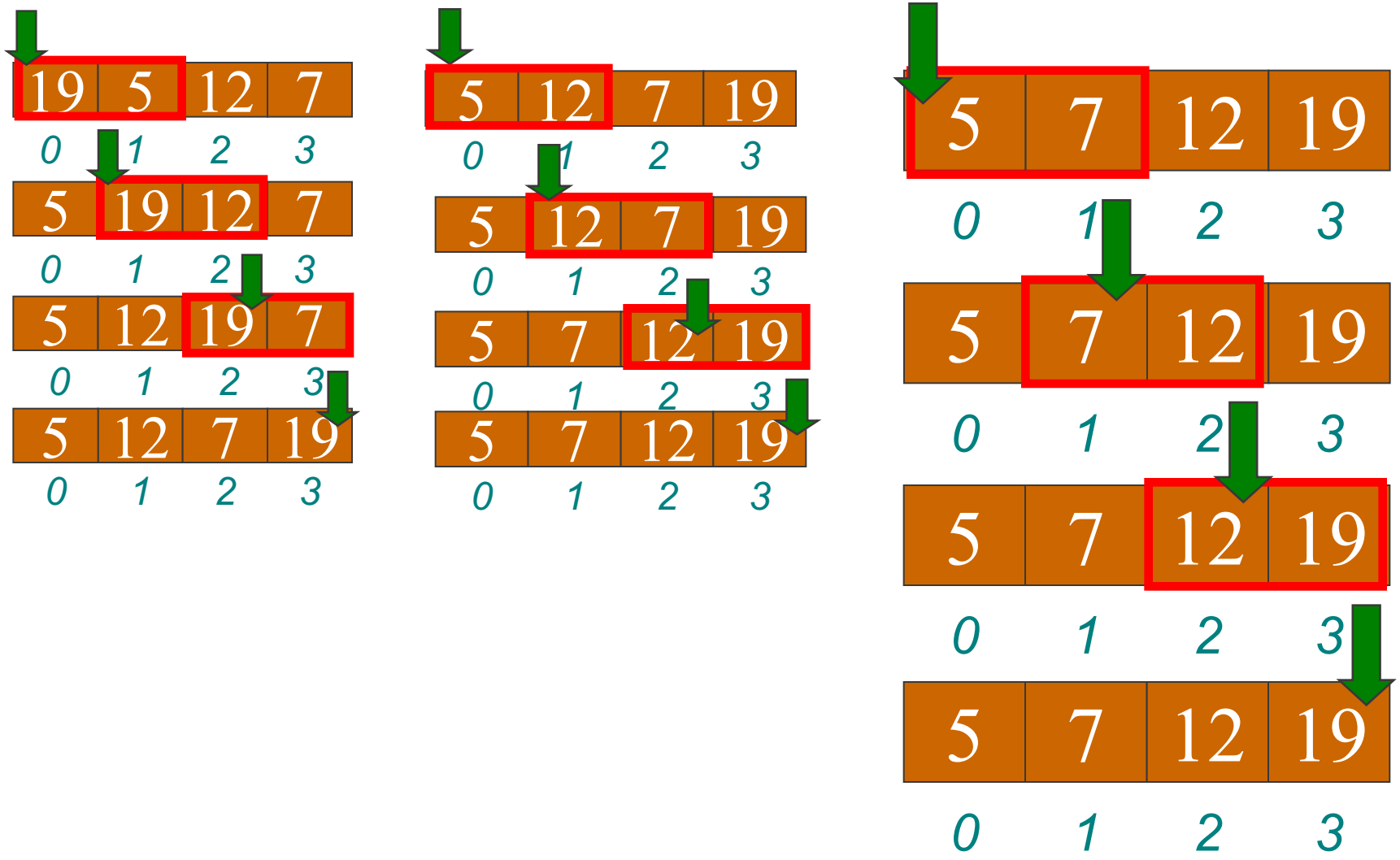


Bubble Sort – Example: `the_list=[19,5,12,7]`

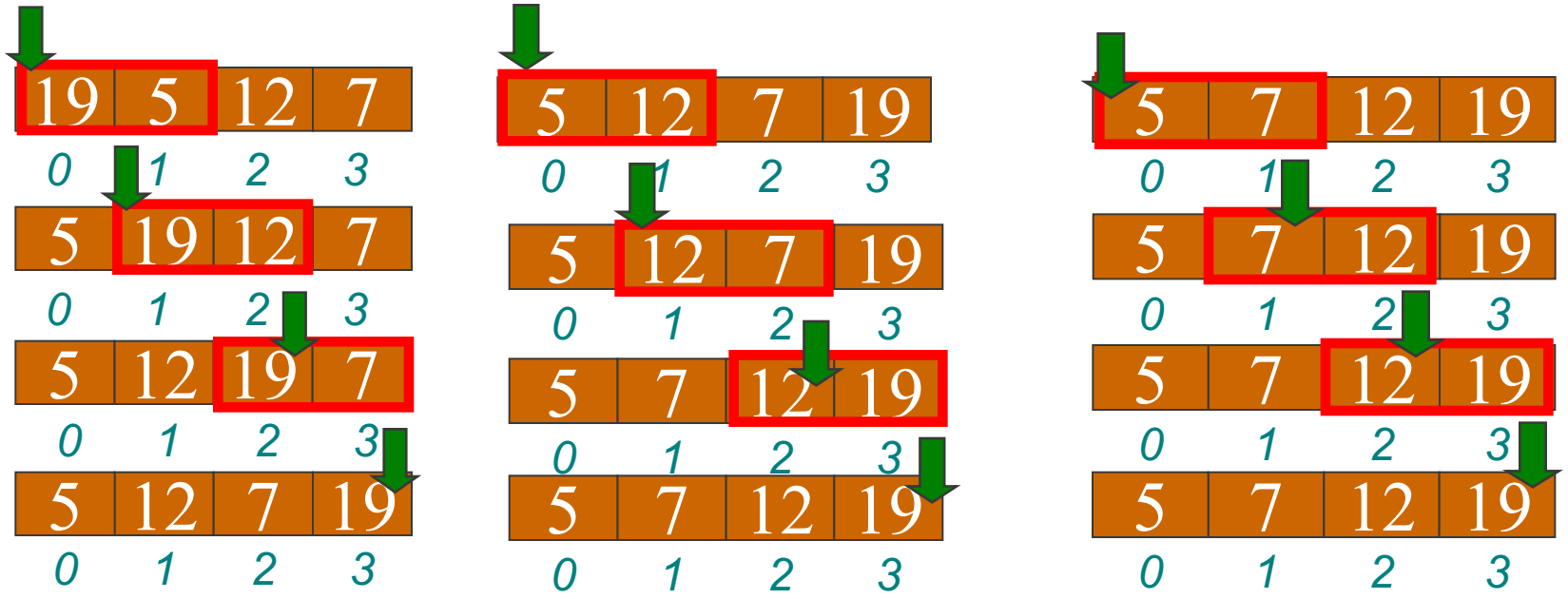




Bubble Sort – Example: `the_list=[19,5,12,7]`



Bubble Sort – Example: `the_list=[19,5,12,7]`



n-1 (3) traversals performed

Bubble Sort: Python Code

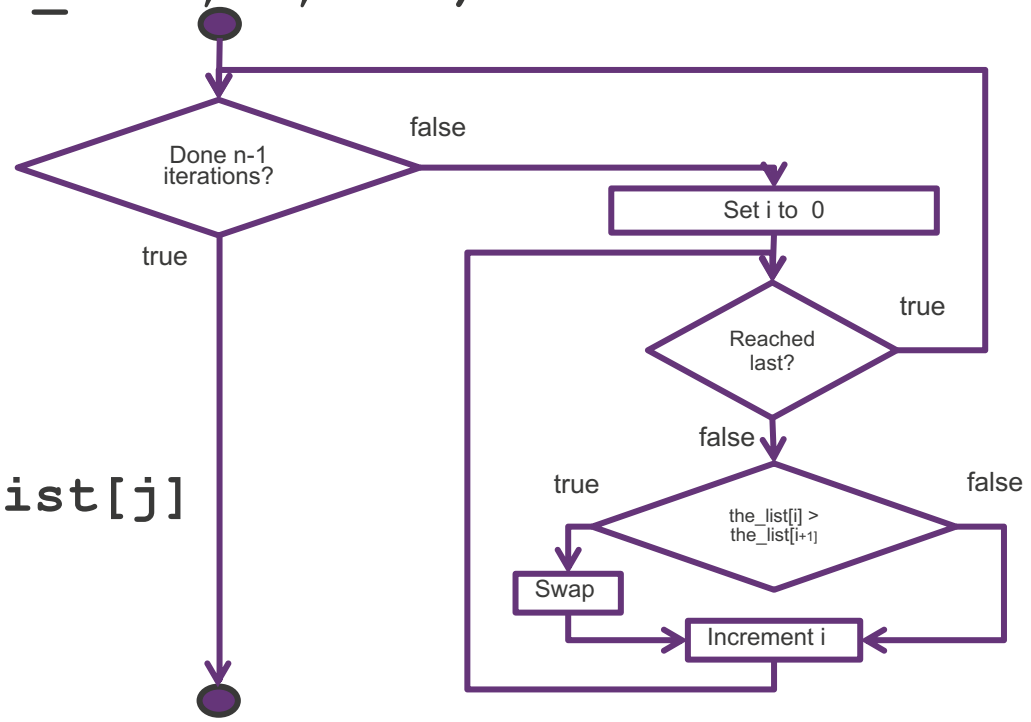
```
def bubble_sort(the_list):
    n = len(the_list)
    for a in range(n-1):
        for i in range(n-1):
            if (the_list[i] > the_list[i+1]):
                swap(the_list, i, i+1)
```

`a` is never used explicitly. Just a way to iterate n-1 times. In Python you can use _ for such variables.

Will not comment code in the slides (no space, and not the focus) but will in the .py code provided with them

The prac code did not define swap, for simplicity. In Python you can swap in a line, but not in many other languages. In those, use a function for reusability. I use it here to illustrate what you should do in those cases.

```
def swap(the_list,i,j):
    tmp = the_list[i]
    the_list[i] = the_list[j]
    the_list[j] = tmp
```



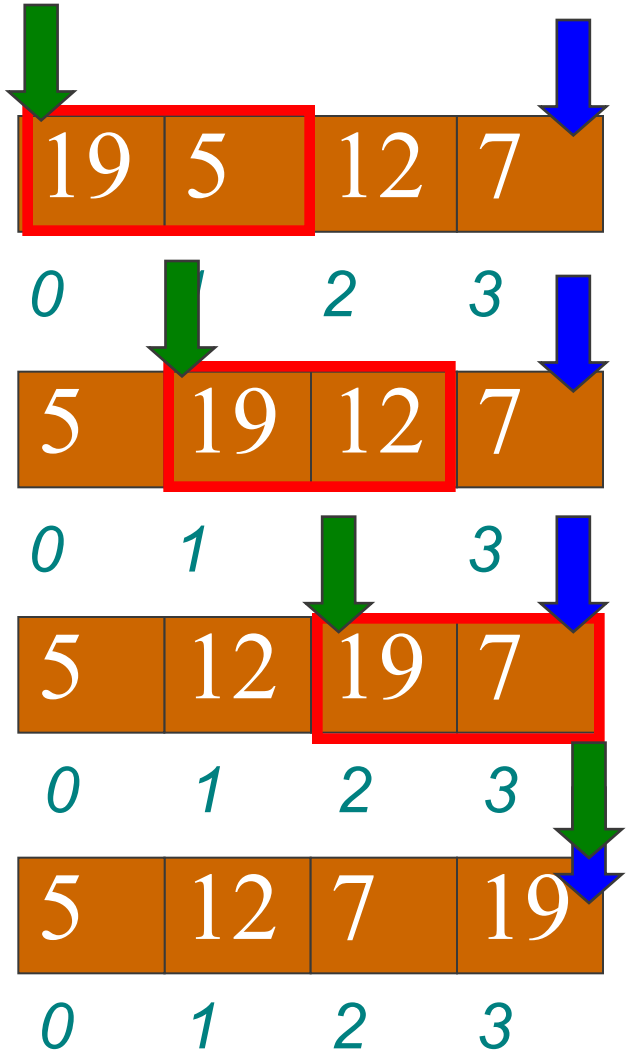
MONASH
University

Bubble Sort Optimisations

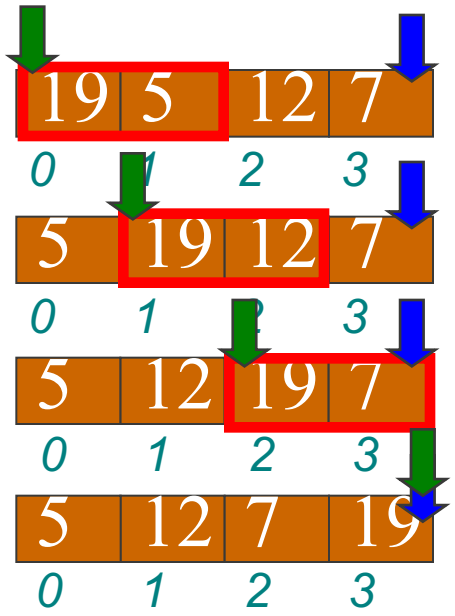
Bubble Sort: Optimization

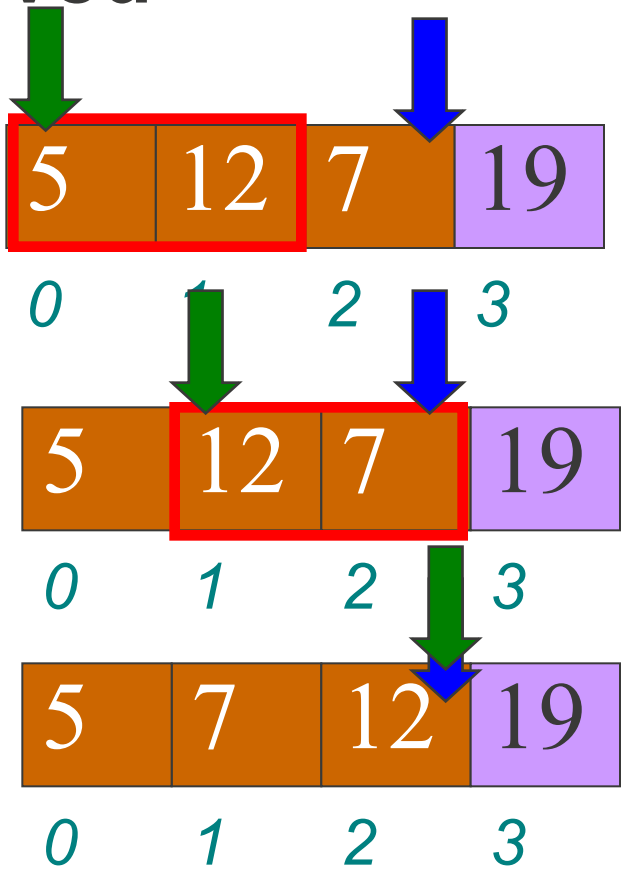
- **We can do better, but how?**
- **Use the invariant:**
  - After every traversal, the largest unsorted element gets to its final place
- **How is that useful?**
  - Each iteration can avoid comparing the last element moved
- **How?**
  - Mark the last element that might need to be moved

Bubble Sort – Example improved

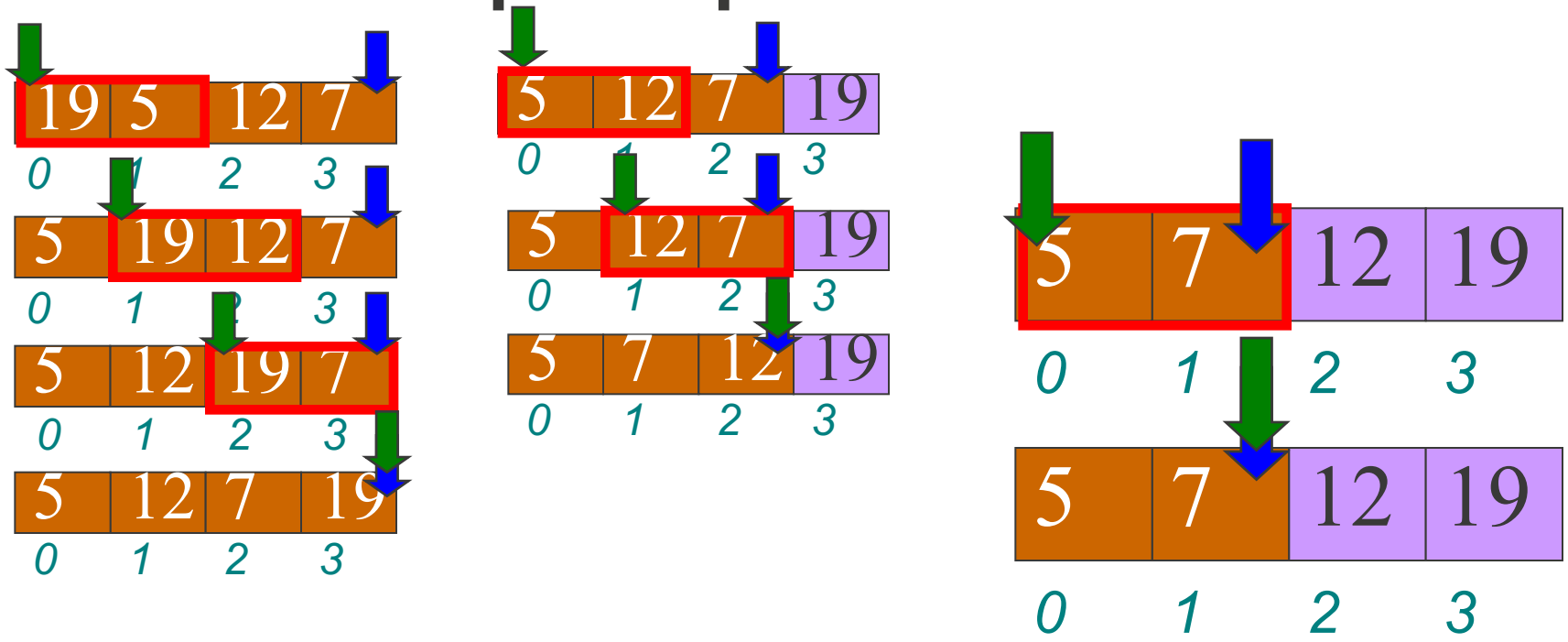


Bubble Sort – Example improved

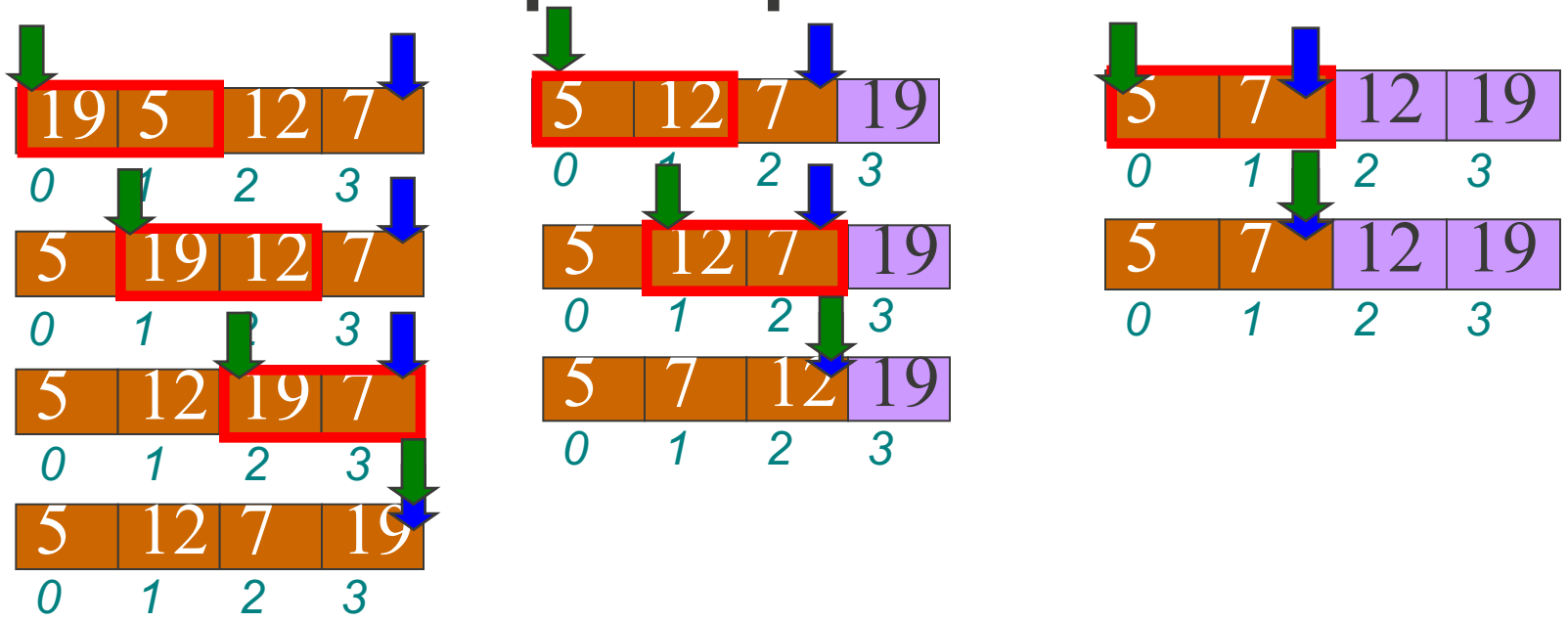




Bubble Sort – Example improved



Bubble Sort – Example improved



In terms of implementation: everything to the right of the (blue) mark is sorted, is in its final position and its size grows by one after each traversal

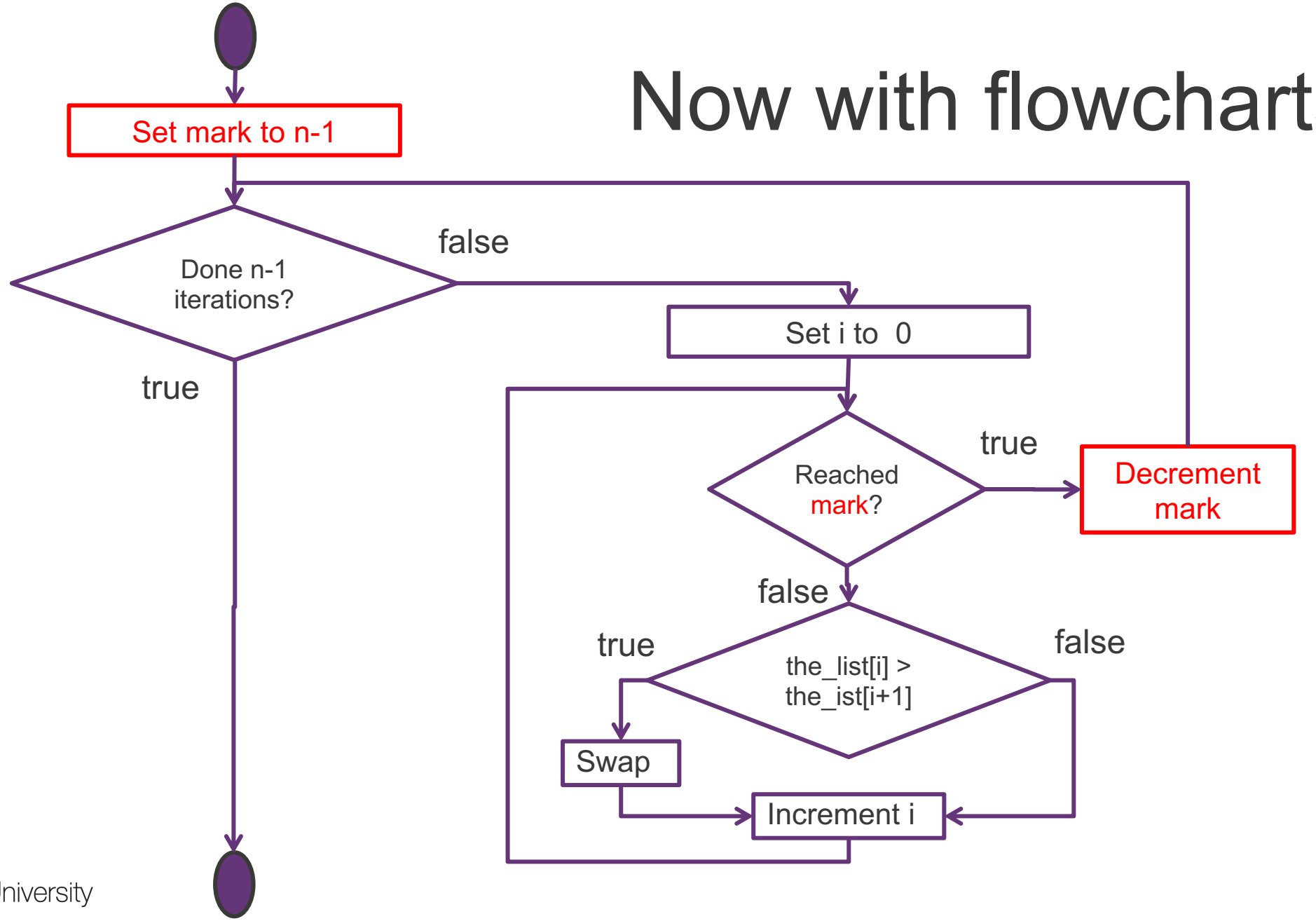
Bubble Sort: one possible algorithm

Set the mark to n-1 New

For n-1 iterations do the following: New

- Start at the leftmost element X
- While we have not reached the mark
  - Compare X to the element Y to its right
  - If X > Y swap them, otherwise don't
  - Move one position to the right
- Decrement the mark New

Now with flowcharts!



Decrementing in a loop

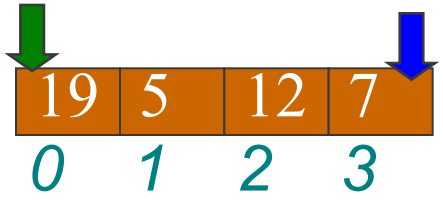
- **The mark needs to go from n-1 to 1 (stop at 0)**
- **How do we do this in Python?**
- **We have seen `range(n)`**
  - Goes from 0 incrementing by 1 until it reaches n
- **Also seen `range(start,stop[,step])`**
  - Goes from **`start`** incre/decrementing to by **`step`**, until it reaches **`stop`**
  - If **`step`** is omitted, its value is 1
  - If **`start`** is omitted, its value is 0
- **So, rather than use `range(n-1)` in our code**
  - We use **`range(n-1,0,-1)`**
  - This will return n-1, n-2, n-3, …, 1

`list()` transforms the sequence returned by range into a list, so I can show you the result in one line

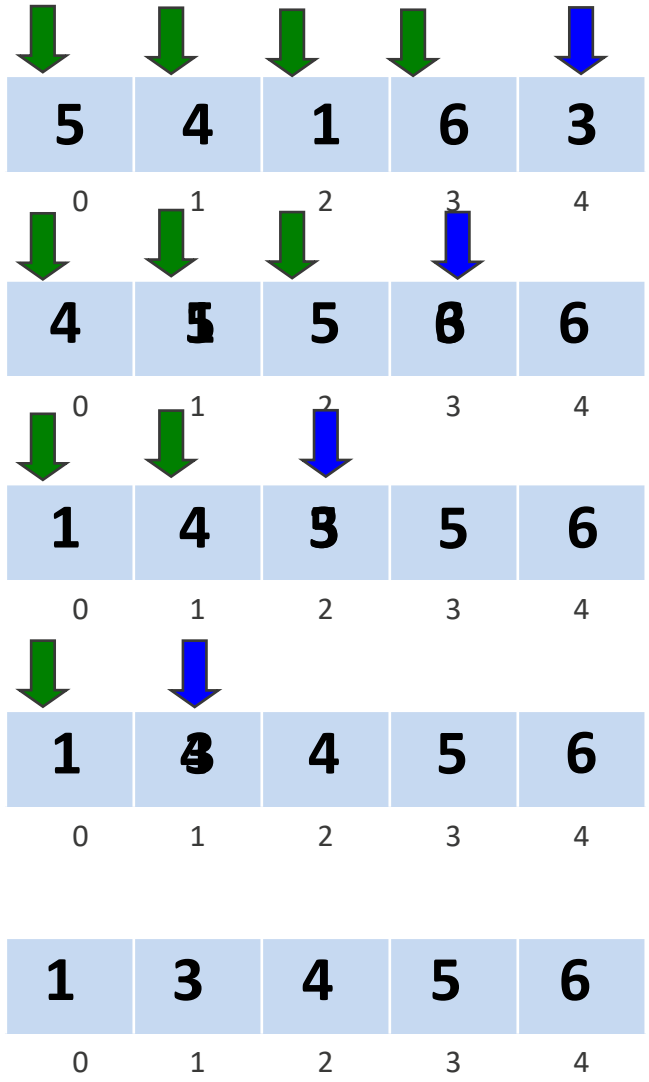
```
>>> list(range(6))
[0,1,2,3,4,5]
>>> list(range(0,6,1))
[0,1,2,3,4,5]
>>> list(range(2,6,1))
[2,3,4,5]
>>> list(range(-2,6,1))
[-2,-1,0,1,2,3,4,5]
>>> list(range(6,0,-2))
[6,4,2]
>>> list(range(6,0))
[]
>>>
```

if only two arguments are provided: they are assumed to be start and stop

Bubble Sort: Python Code




```
def bubble_sort(the_list):
    n = len(the_list)
    for mark in range(n-1,0,-1):
        for i in range(mark):
            if (the_list[i] > the_list[i+1]):
                swap(the_list, i, i+1)
```




```
def bubble_sort(the_list):
    n = len(the_list)
    for mark in range(n-1,0,-1):
        for i in range(mark):
            if (the_list[i]>the_list[i+1]):
                swap(the_list, i, i+1)
```

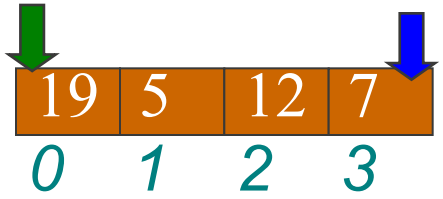
Bubble Sort: More optimizations

- **Consider the list of elements:**

- **What happens after 1 iteration?**

- **It is sorted! What happens in the next iteration?**
- **No swaps! But we still run all n-1 iterations!**
- **How can we take advantage of this?**
- **Detect it. Use a boolean `swapped` initialised to false**
  - Set to true every time there is a swap
  - If after one iteration not swapped is true: stop

Bubble Sort: Python Code




```
def bubble_sort(the_list):
    n = len(the_list)
    for mark in range(n-1,0,-1):
        swapped = False
        for i in range(mark):
            if (the_list[i] > the_list[i+1]):
                swap(the_list, i, i+1)
                swapped = True
        if not swapped:
            break
```

We will call this algorithm BubbleSort II

Breaks out the closest enclosing **`for`** or **`while`** loop. Usually not a good idea, but very useful when at the end (simulates to a do-while loop, which some of you might know).

MONASH University

Stability and Incrementality

Is our Bubble Sort algorithm incremental?

- **An algorithm is incremental if it does not need to re-compute everything after a small change**
  - Can reuse most of the work already done to handle the change
- **A sorting algorithm is incremental if it can:**
  - Given a sorted list and one new element
  - Use one (or a few) iterations of the algorithm to return a sorted list that has the new element
- **Consider the sorted list**

- **If we now receive element 13, can bubble sort handle it incrementally?**

Is this algorithm incremental? (cont)

- **If we append 13 at the end**

3	6	10	14	18	20	13

- **How many iterations are needed until it is sorted?**
  - 1st

3	6	10	14	18	13	20

  - 2nd

3	6	10	14	13	18	20

  - 3rd

3	6	10	13	14	18	20

  - 4th: detect no swaps and finish
- **Not very incremental**

Why? It is just a few iterations…

  - If the new element is the smallest: runs all iterations (n-1)
  - Number of extra iterations needed is not a small constant (depends on position)

Is this algorithm incremental? (cont)

- **If we add 13 at the beginning**

- **How many iterations are needed until it is sorted?**
  - 1st

  - 2nd: detect no swaps and finish

This works for Bubble Sort II only!

- **Very incremental**
  - We can guarantee that after one iteration it is always sorted
- **But how much work is it to add it to the beginning?**
  - As we will see, we then need to shuffle everything to the right
  - This takes one iteration

So a total of two iterations always

  - Adding at the end was faster, but sorting then could require all iterations

Properties of sorting algorithms

Algorithm	Stable	Incremental
Bubble Sort		No
Bubble Sort II		Yes (add to front)

Is this algorithm stable?

- **A sorting algorithm is stable if it:**
  - Maintains the relative order among elements
- **Example: given the list**

8	3	8	6	3
a	b	c	d	e

- **As stable sort will always obtain**
  - The relative order is preserved
  - That is: b before e, a before c

3	3	6	8	8
b	e	d	a	c
✓			✓	

- **A non stable sort might obtain**
  - Changing relative order of b and e

3	3	6	8	8
e	b	d	a	c
✗			✓	

Name	Mark
Ann	100
Brendon	90
Cheng	100
Daniel	50

Name	Mark
Daniel	50
Brendon	90
Cheng	100
Ann	100

Cheng before Ann

Not stable

Name	Mark
Daniel	50
Brendon	90
Ann	100
Cheng	100

Ann before Cheng

Stable

Is Bubble Sort stable?

The only time in which elements can get out of order is when we swap

But we only swap when `the_list[i]` is > (never equal) to `the_list[i+1]`

```
def bubble_sort(the_list):
    n = len(the_list)
    for mark in range(n-1,0,-1):
        for i in range(mark):
            if (the_list[i] > the_list[i+1]):
                swap(the_list, i, i+1)
```

make sure inequality is strict

So, yes, it is stable

8	3	8	6	3
a	b	c	d	e

Can we ensure a and b are always before c and e, respectively?

Yes, but careful, a small change (>= rather than >) makes it non stable

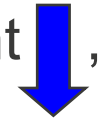
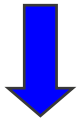
Properties of sorting algorithms

Algorithm	Stable	Incremental
Bubble Sort	Yes (strict)	No
Bubble Sort II	Yes (strict)	Yes (add to front)

MONASH
University

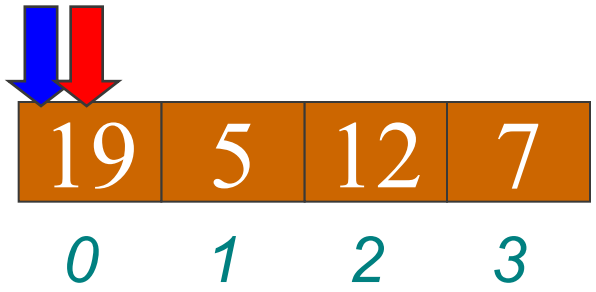
Selection Sort

Selection Sort

- **Main idea: no need to perform so many swaps**
- **In every iteration:**
  - Start at the leftmost unsorted element  , set it as the current minimum
  - Traverse the rest and find the minimum element in the rest of the list (if different from current)
  - Swap it with the leftmost unsorted element 

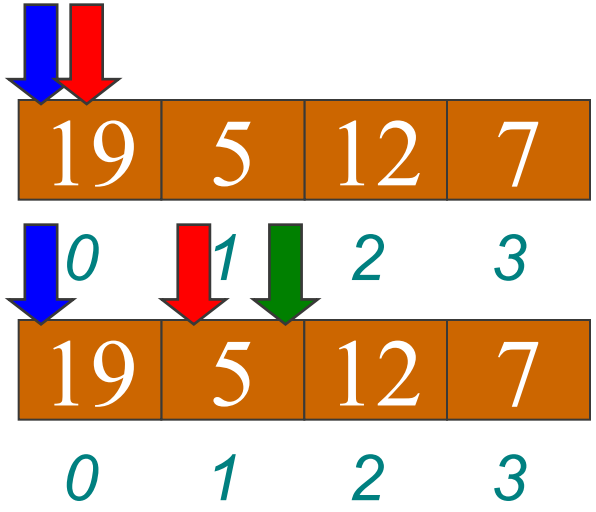
Selection Sort – Example

- Start at leftmost unsorted
- Traverse rest to find min
- Swap it with leftmost unsorted



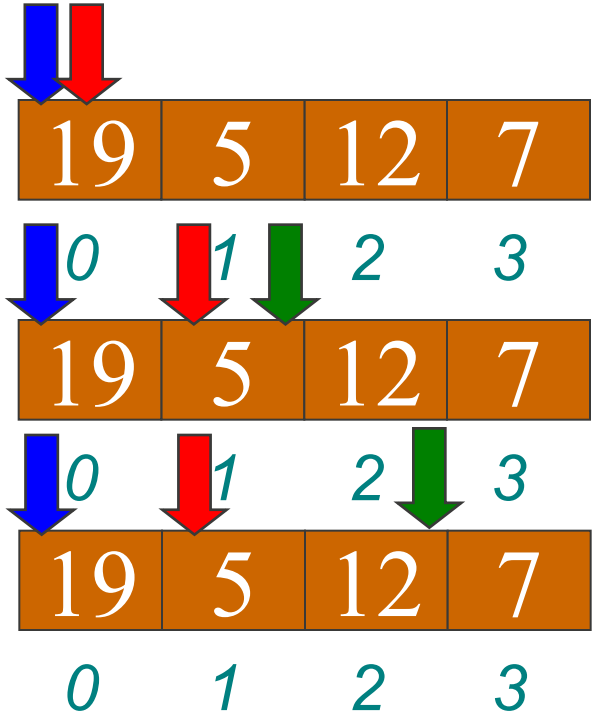
Selection Sort – Example

- Start at leftmost unsorted
- Traverse rest to find min
- Swap it with leftmost unsorted



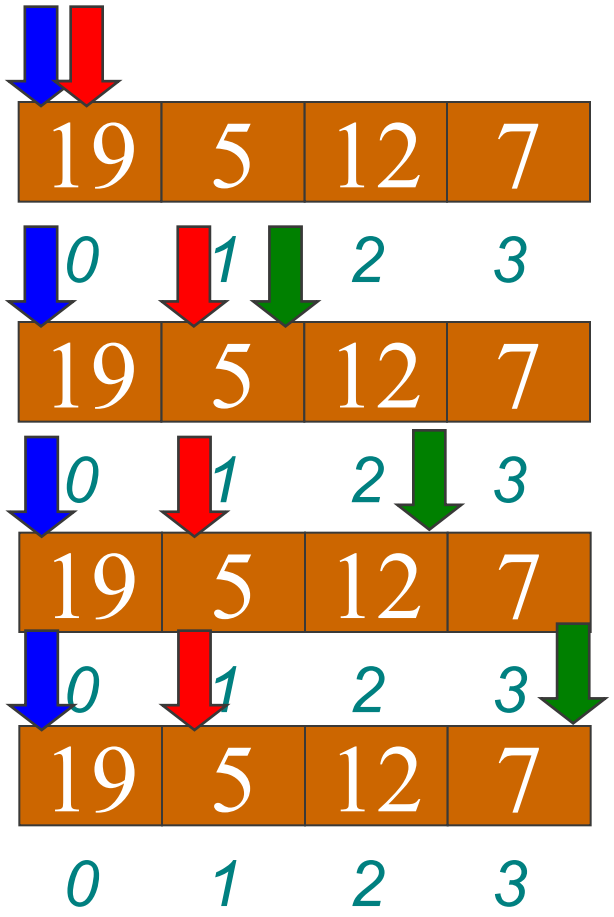
Selection Sort – Example

- Start at leftmost unsorted
- Traverse rest to find min
- Swap it with leftmost unsorted



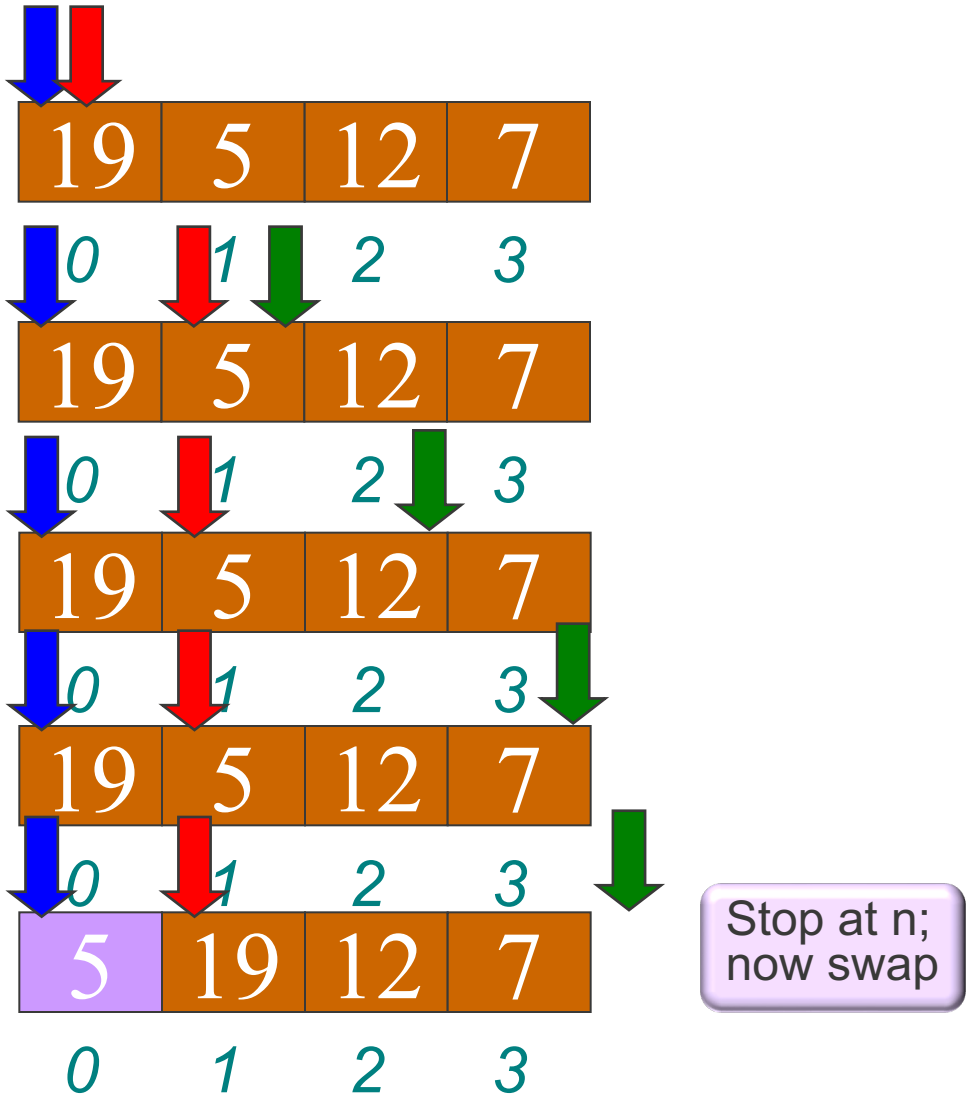
Selection Sort – Example

- Start at leftmost unsorted
- Traverse rest to find min
- Swap it with leftmost unsorted

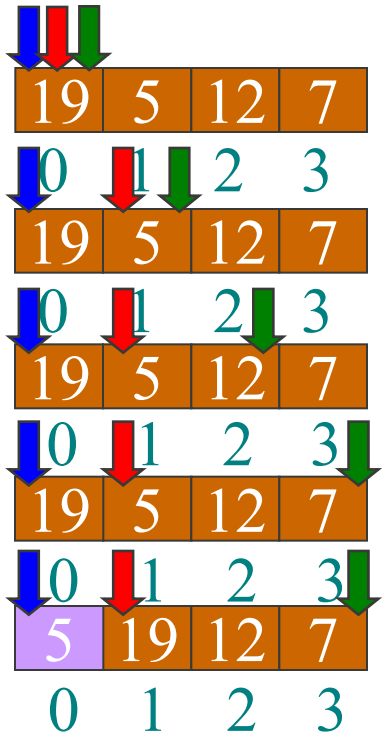


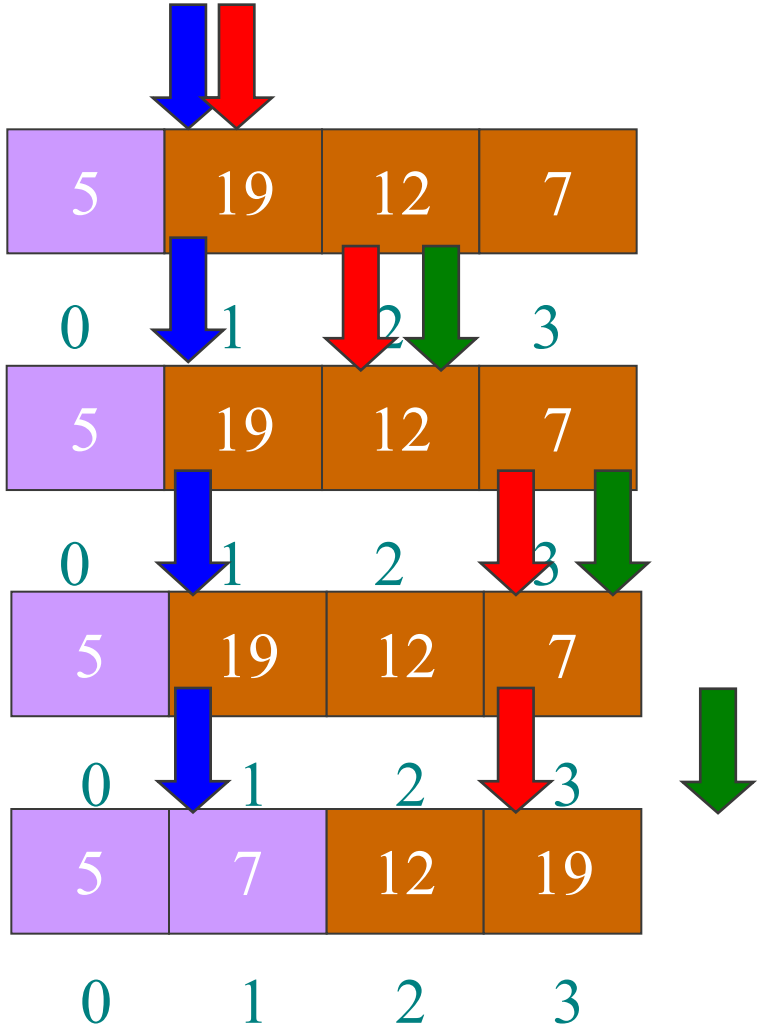
Selection Sort – Example

- Start at leftmost unsorted
- Traverse rest to find min
- Swap it with leftmost unsorted

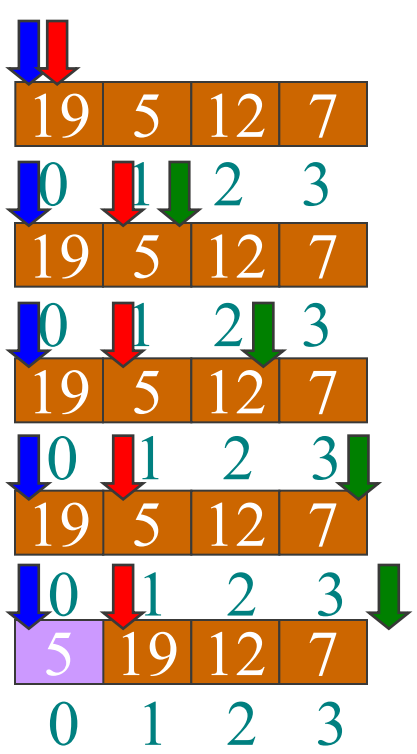


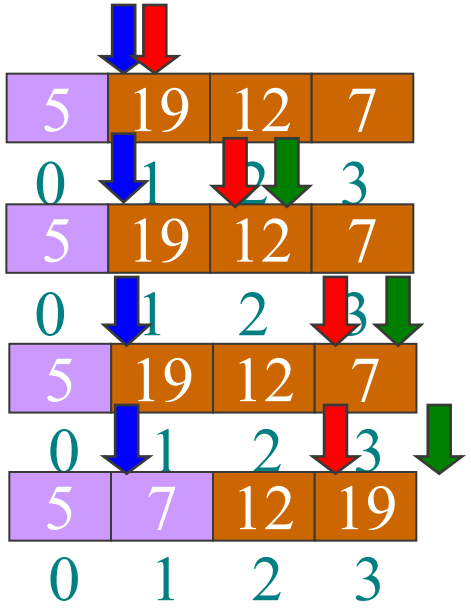
Selection Sort – Example

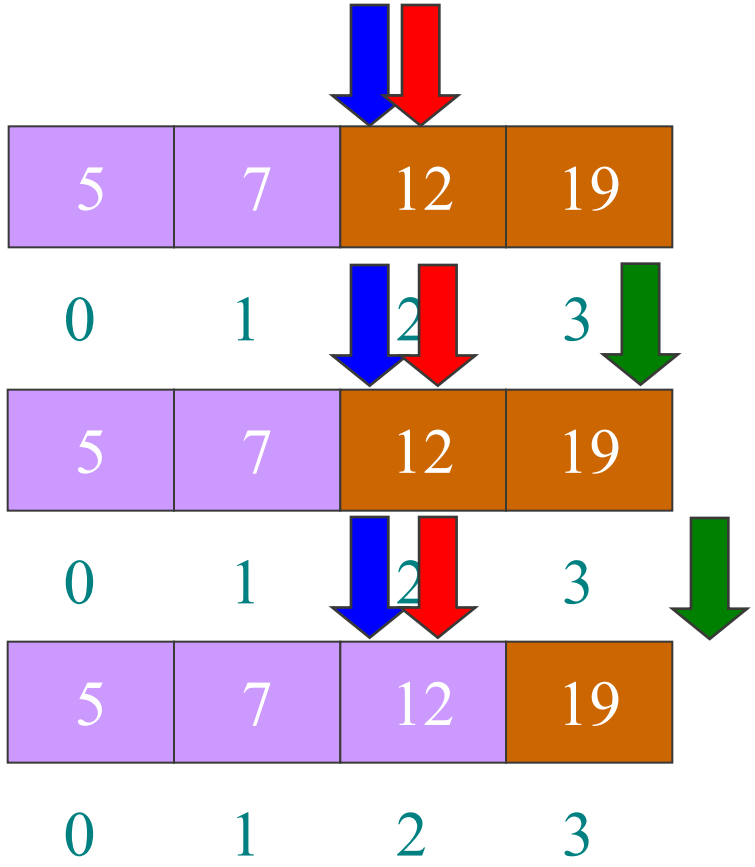




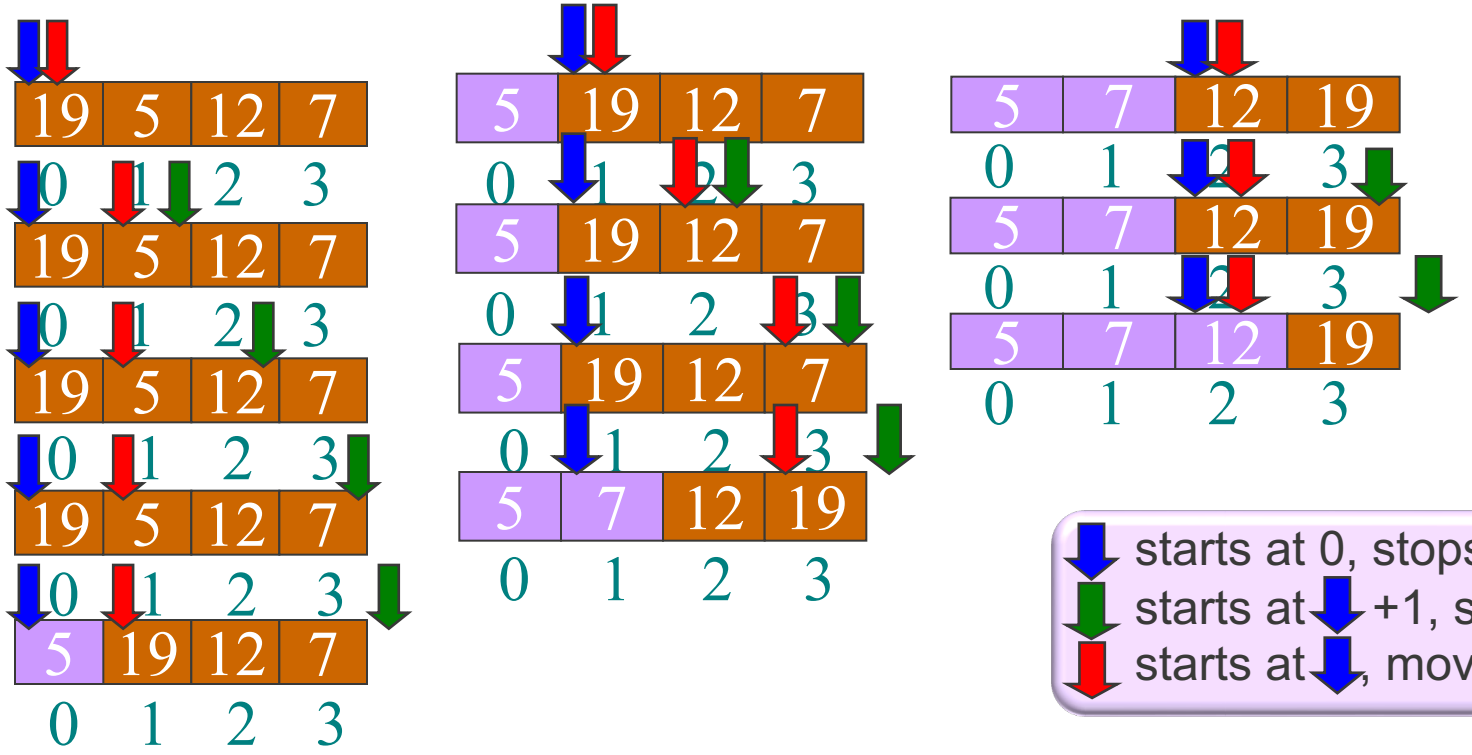
Selection Sort – Example







Selection Sort – Example

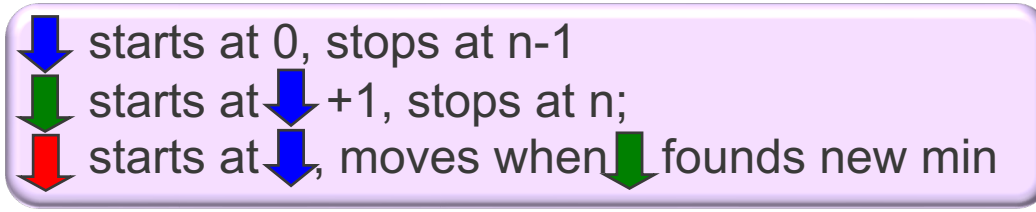


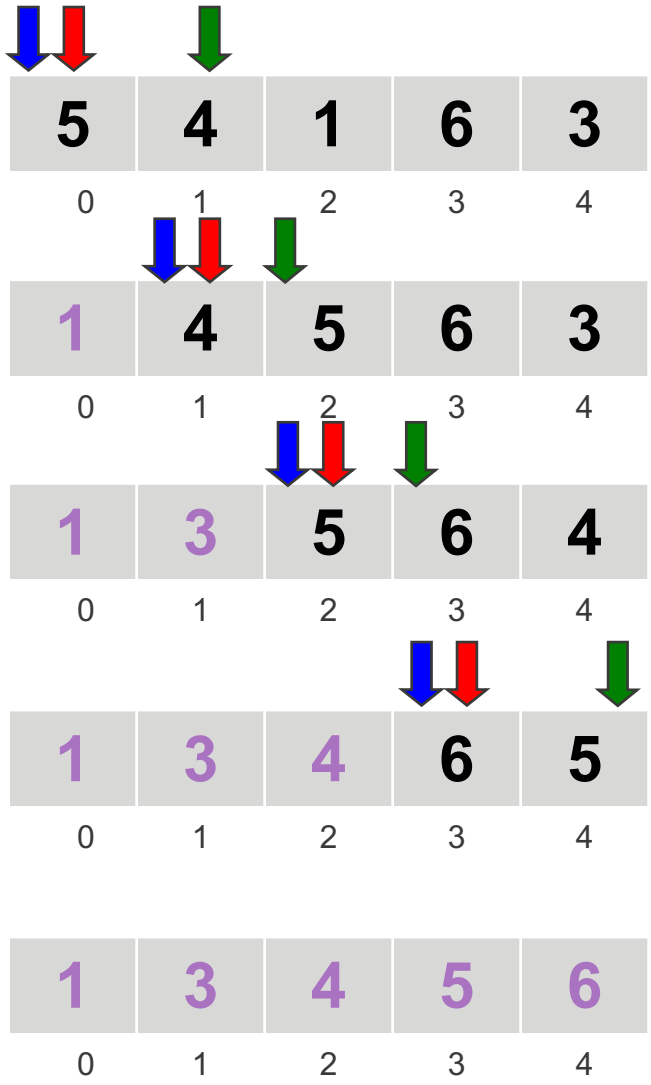
In terms of implementation: everything to the left of the (blue) mark is sorted, is in its final position and grows by one after each traversal

Selection Sort: Code

```
def selection_sort(the_list):
    n = len(the_list)
    for mark in range(n-1):
        min_index = find_min(the_list,mark)
        swap(the_list, mark, min_index)


def find_min(the_list,mark):
    pos_min = mark
    n = len(the_list)
    for i in range(mark+1,n):
        if the_list[i]<the_list[pos_min]:
            pos_min = i
    return pos_min
```






```
def selection_sort(the_list):
    n = len(the_list)
    for mark in range(n-1):
        min_index = find_min(the_list,mark)
        swap(the_list, mark, min_index)


def find_min(the_list,mark):
    pos_min = mark
    n = len(the_list)
    for i in range(mark+1,n):
        if the_list[i]<the_list[pos_min]:
            pos_min = i
    return pos_min
```

Selection Sort: going deeper

- **Can we detect that we are already sorted?**
- **In Bubble Sort we did this with a boolean variable**
  - `swapped`
- **Can we do something similar here?**
  - In each iteration: we are looking for the minimum
  - This tell us nothing about the relative order of elements
  - We cannot use that information to stop

Properties of sorting algorithms

Algorithm	Stable	Incremental
Bubble Sort	Yes (strict)	No
Bubble Sort II	Yes (strict)	Yes (add to front)
Selection Sort		

Is this Selection Sort incremental?

- **Consider again the sorted list**

- **If we now receive element 13, can Selection Sort handle it incrementally?**
- **If we appended to the end:**
  - The list will remain unchanged for the first 3 iterations
  - Even when the mark arrives to the correct position for 13 we have not finished (number 14 is now at the end of the list!)
  - And even if we had finished, our algorithm would not realise that!
  - Could we have used the mark to help? (start with mark at 20)

    - No, that would be wrong (assumes pink elements are in final position!)
- **Since it needs to do all the work again, it is not incremental**

Is this Selection Sort stable?

Using colour to differentiate the two 14s

- Consider again the sorted list

- In the first three iterations nothing happens
- In the fourth, we swap the white 14 and the 13 ending up as

- In the remaining iterations nothing changes
- The two 14s have swapped!
- But the swap was done after a strict > comparison! How did this happen?
- We are swapping non-consecutive elements!
- So not stable

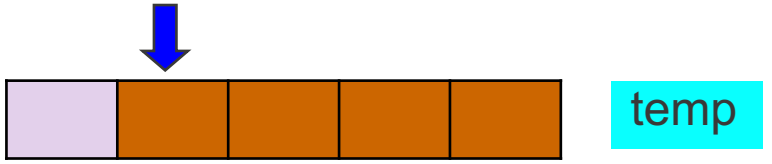
Properties of sorting algorithms

Algorithm	Stable	Incremental
Bubble Sort	Yes (strict)	No
Bubble Sort II	Yes (strict)	Yes (add to front)
Selection Sort	No	No

MONASH University

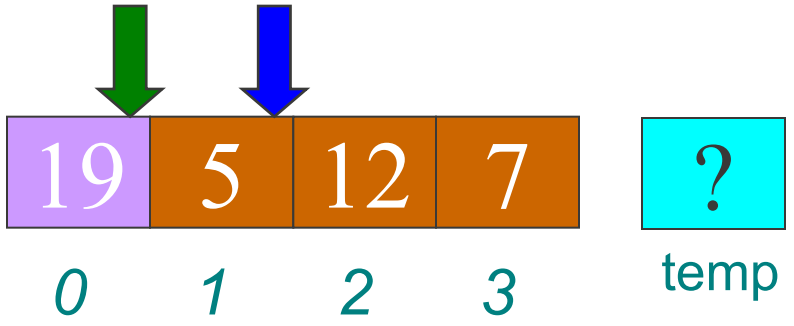
Insertion Sort

Insertion Sort



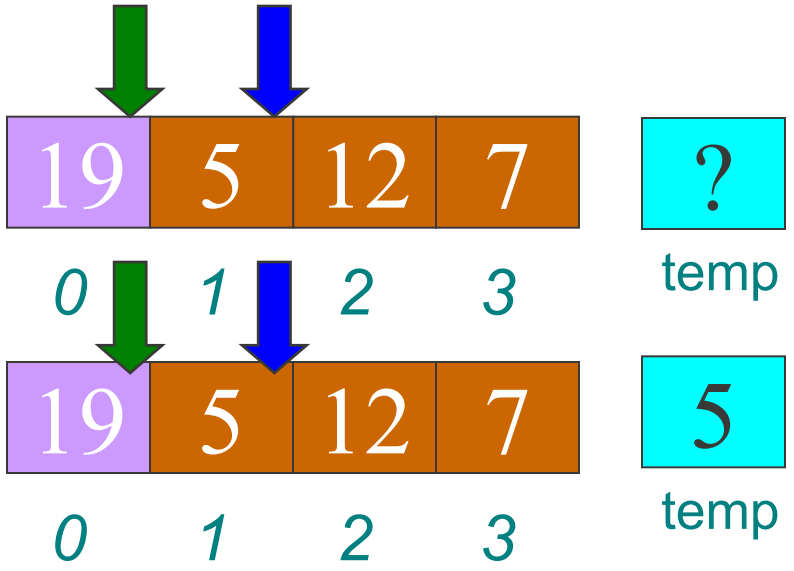
- **Main idea:**
  - Split the list into
    - Part S which is already sorted (initially one element)
    - Part U which is unsorted
  - Extend S by taking any element from U and inserting it in S maintaining the order
- **Invariant: elements in S (left of) are sorted and grow by 1 in each iteration**
- **So the number of iterations required is again n-1**
  - Careful: elements in S might not yet be in their final position
  - Others may move in between them later
- **For every n-1 iteration:**
  - Store the first unsorted value X in a temporary place
  - Shift all sorted elements bigger than X, one position to the right
  - Copy X into the newly freed space

Insertion Sort – Example



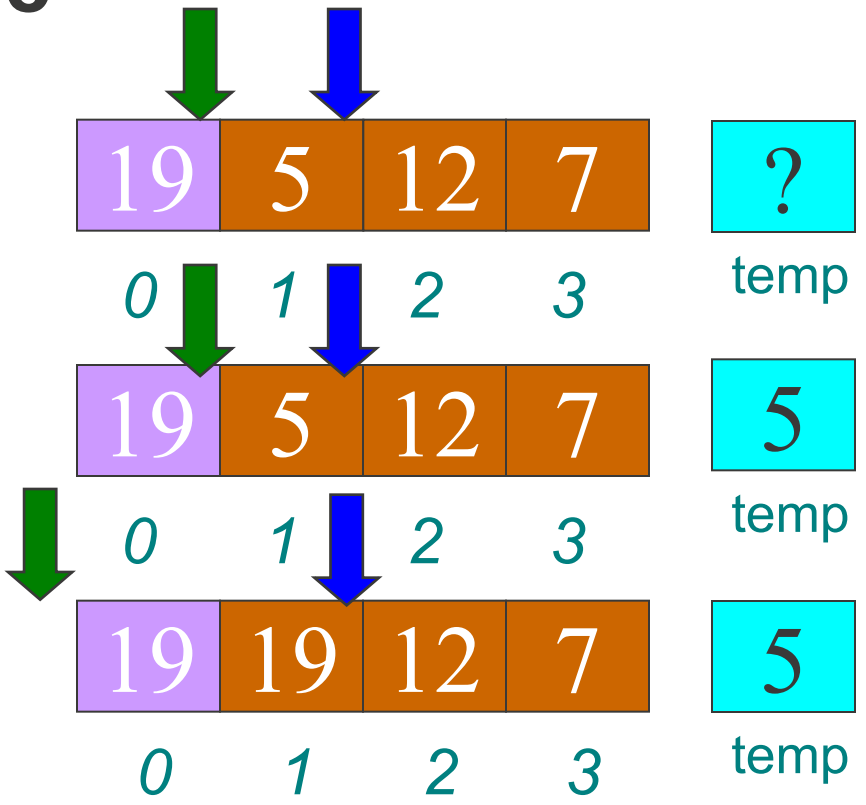
- **Store unsorted in temp**
- **Shift bigger to right**
- **Store temp into freed**

Insertion Sort – Example



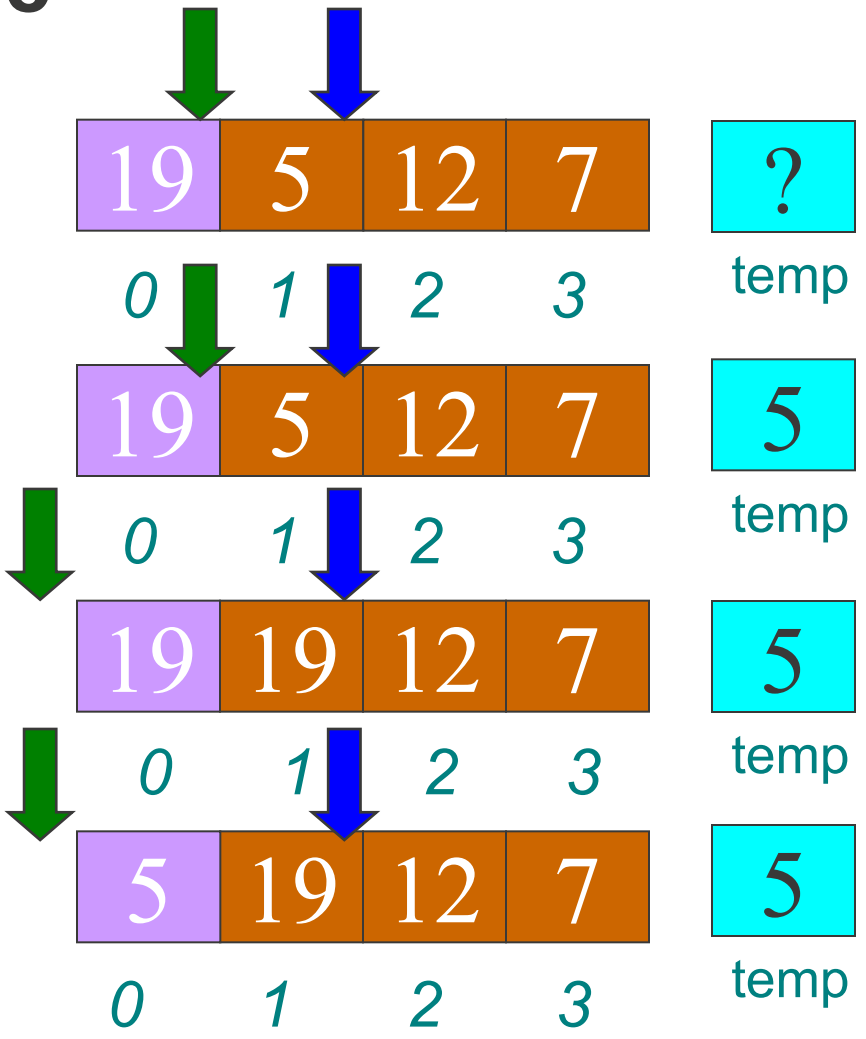
- Store unsorted in temp
- Shift bigger to right
- Store temp into freed

Insertion Sort – Example



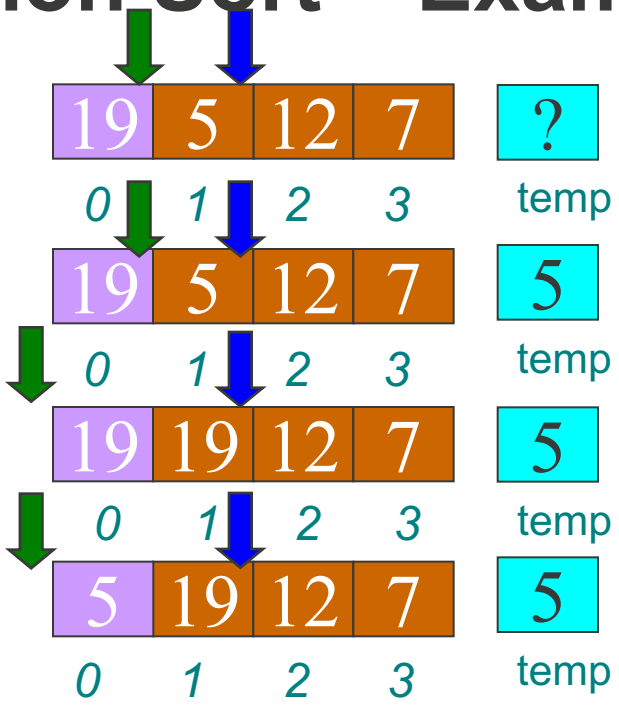
- Store unsorted in temp
- Shift bigger to right
- Store temp into freed

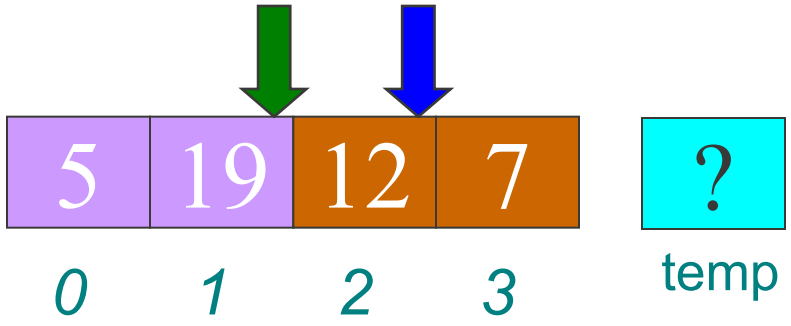
Insertion Sort – Example



- Store unsorted in temp
- Shift bigger to right
- Store temp into freed

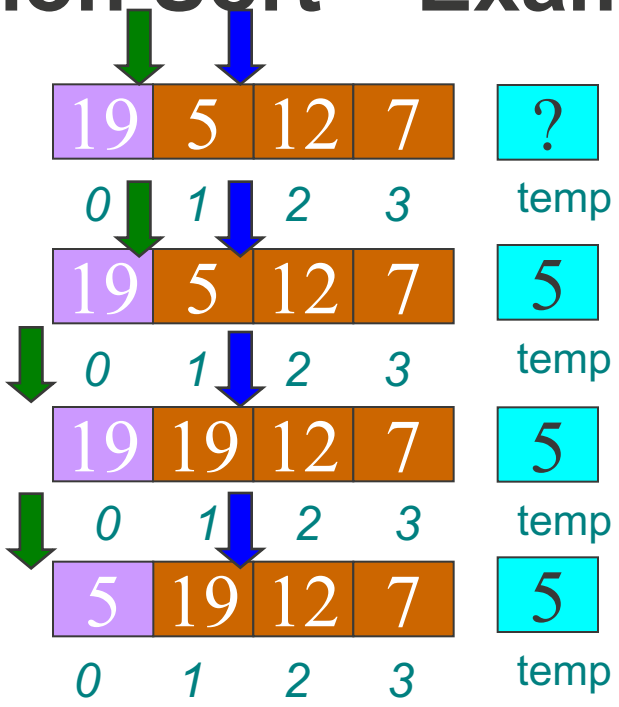
Insertion Sort – Example

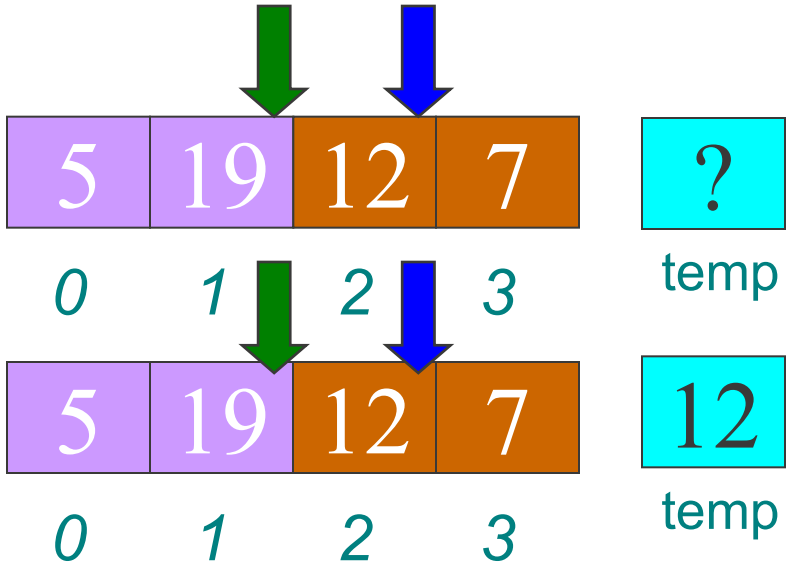




- **Store unsorted in temp**
- **Shift bigger to right**
- **Store temp into freed**

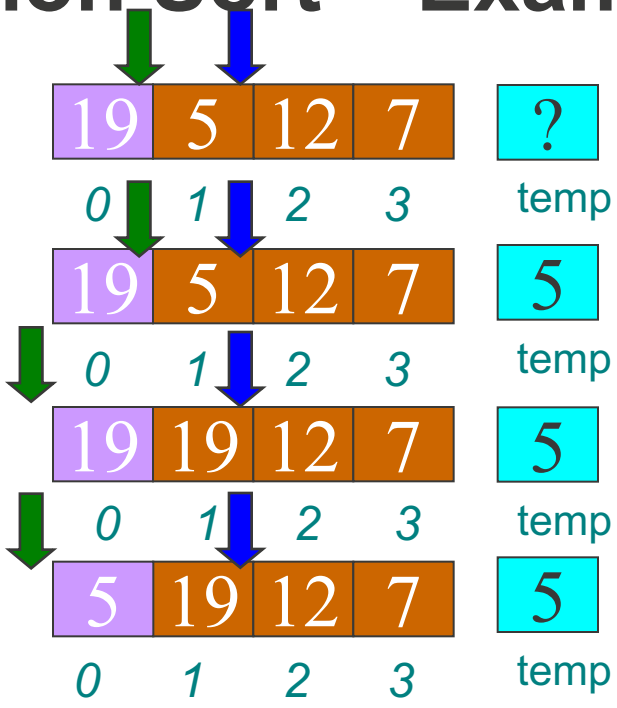
Insertion Sort – Example

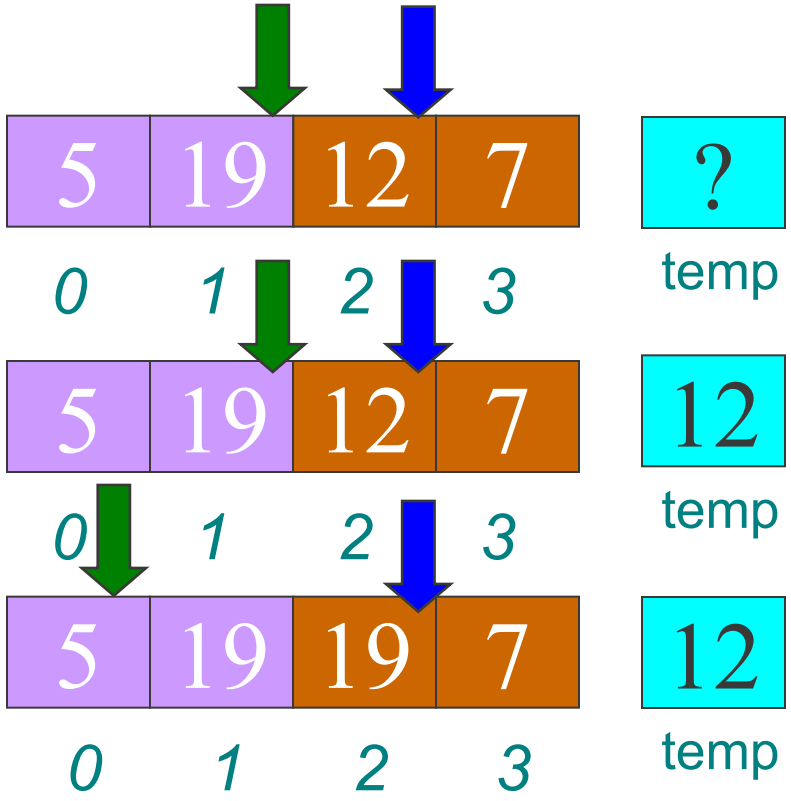




- **Store unsorted in temp**
- **Shift bigger to right**
- **Store temp into freed**

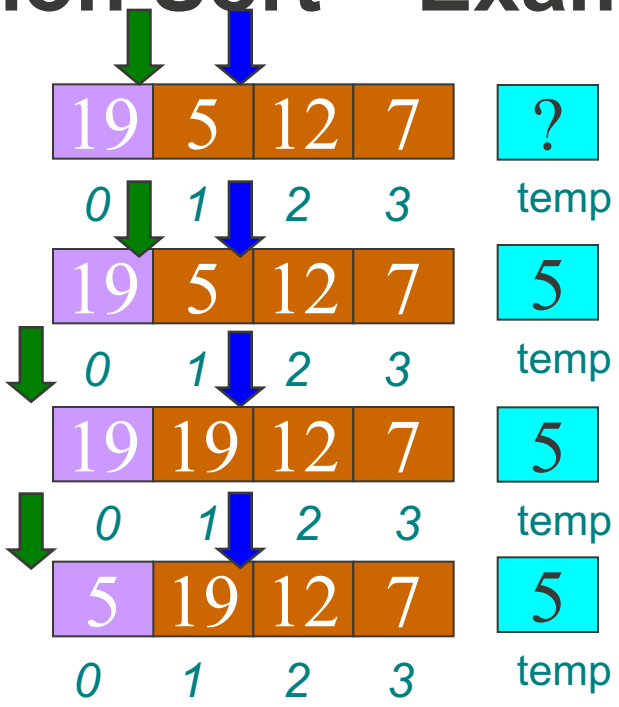
Insertion Sort – Example

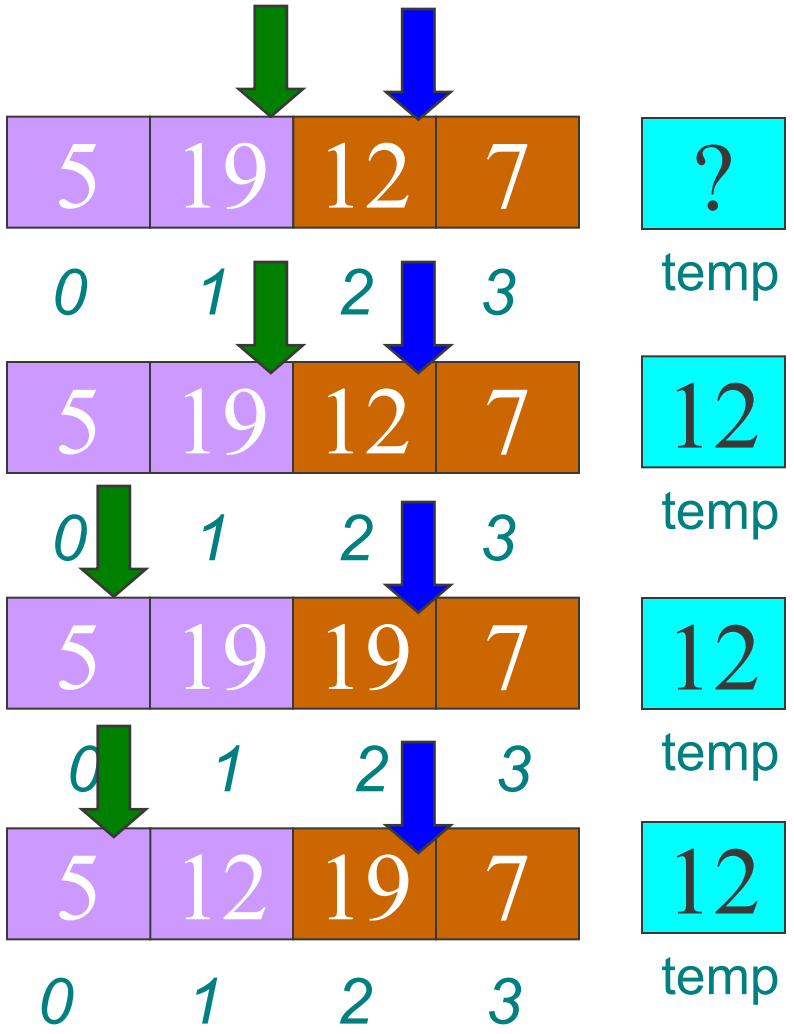




- **Store unsorted in temp**
- **Shift bigger to right**
- **Store temp into freed**

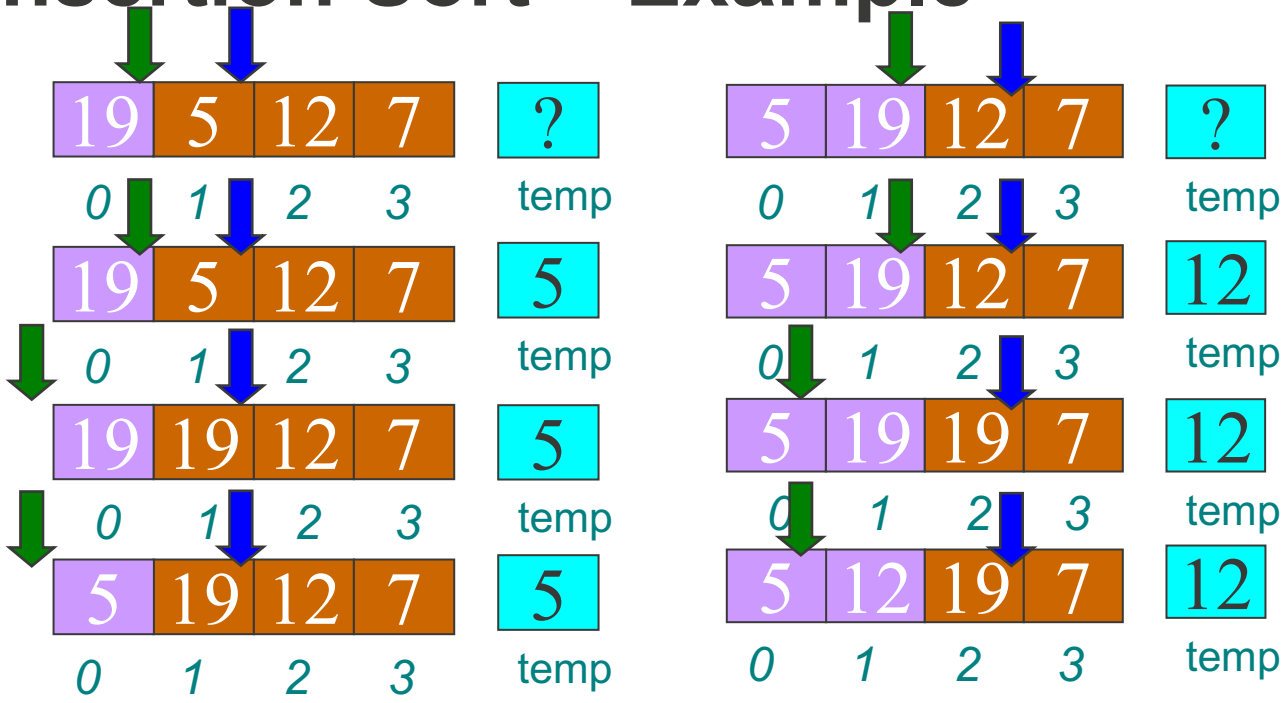
Insertion Sort – Example

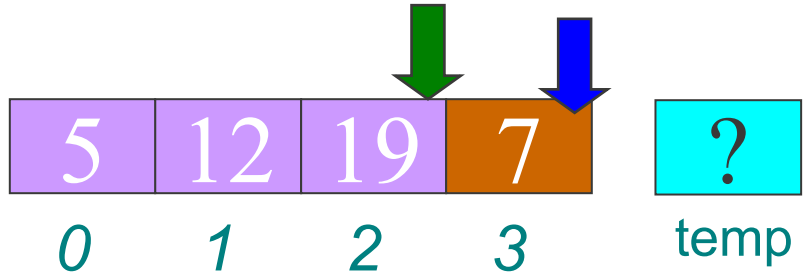




- **Store unsorted in temp**
- **Shift bigger to right**
- **Store temp into freed**

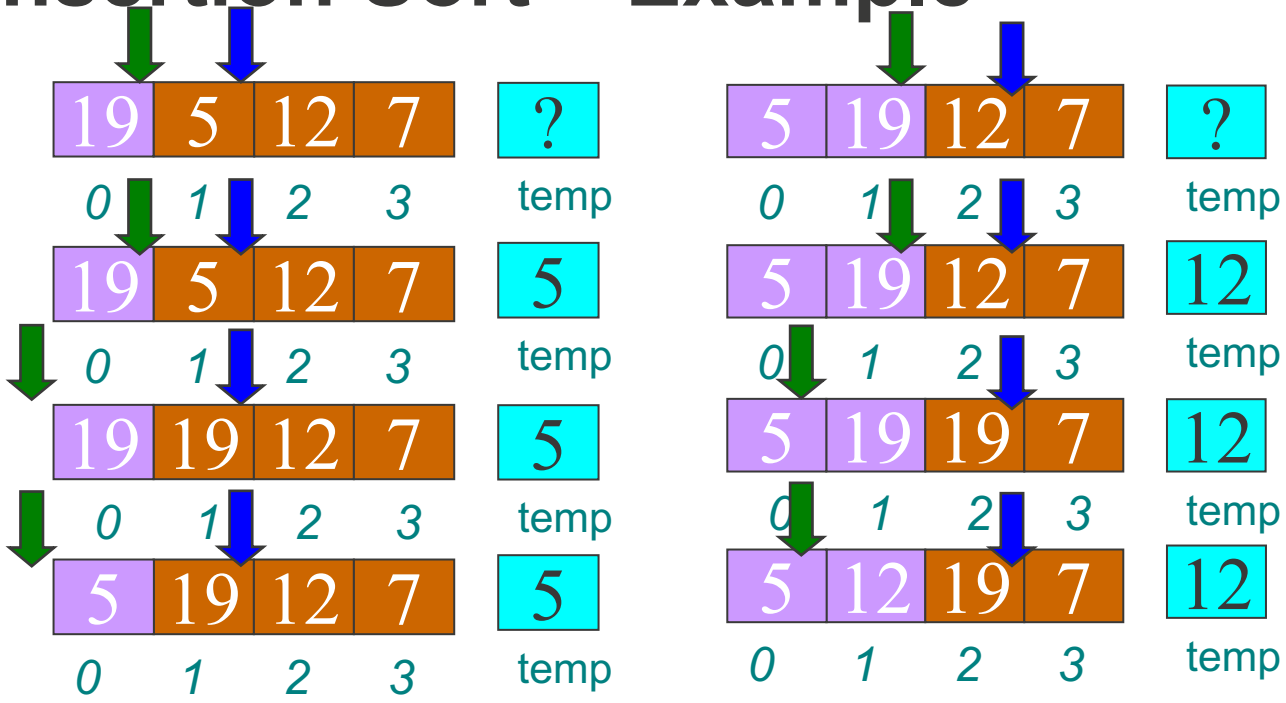
Insertion Sort – Example

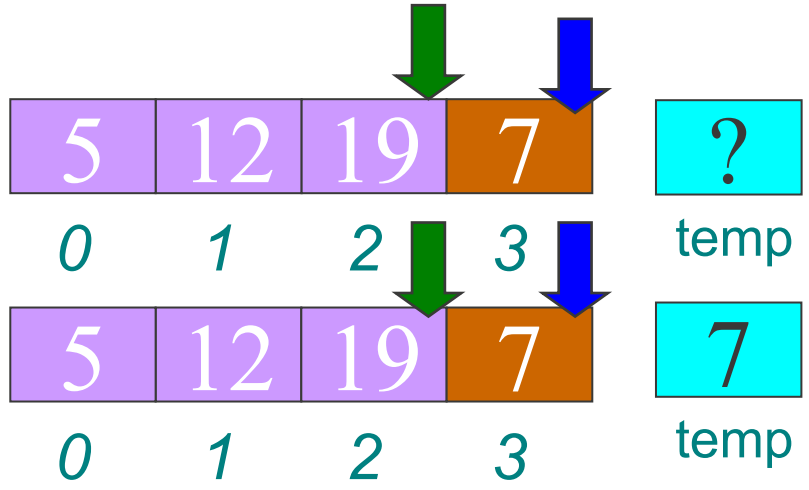




- **Store unsorted in temp**
- **Shift bigger to right**
- **Store temp into freed**

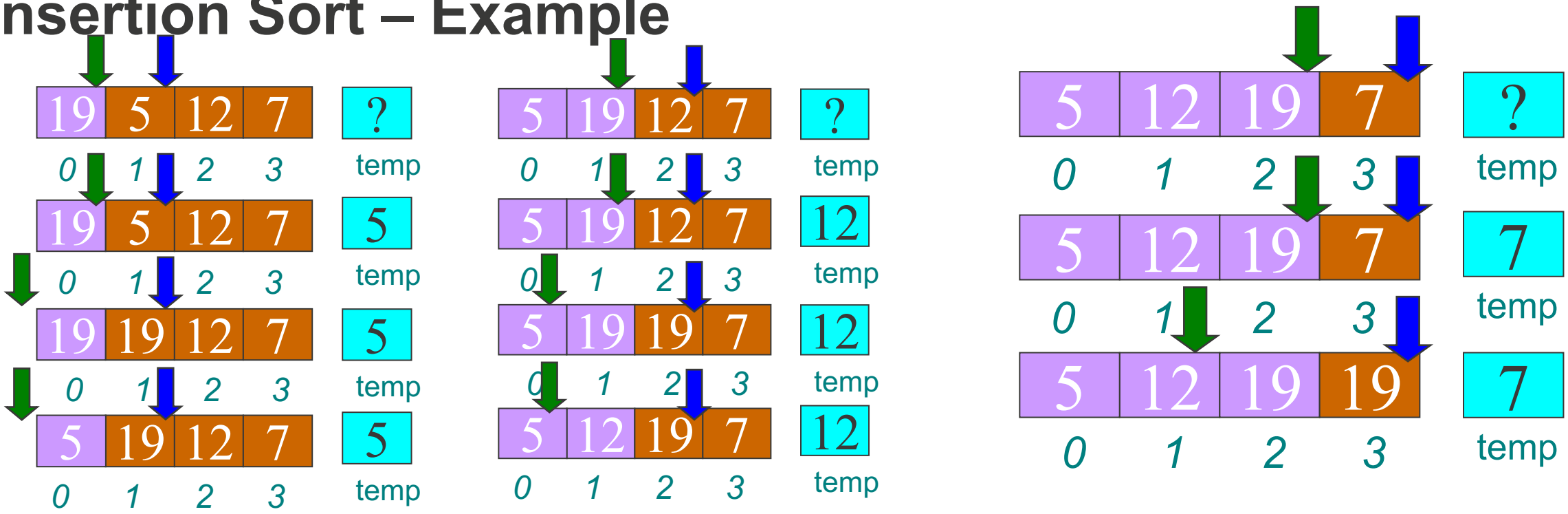
Insertion Sort – Example





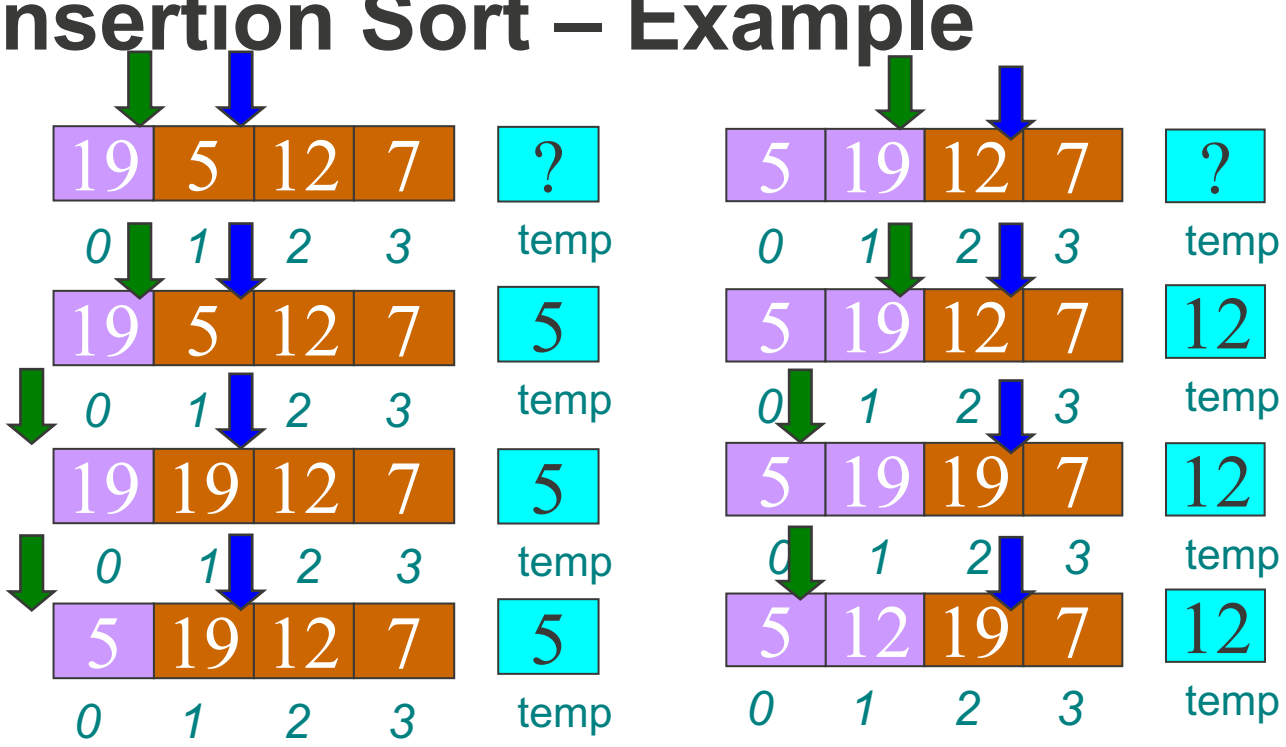
- **Store unsorted in temp**
- **Shift bigger to right**
- **Store temp into freed**

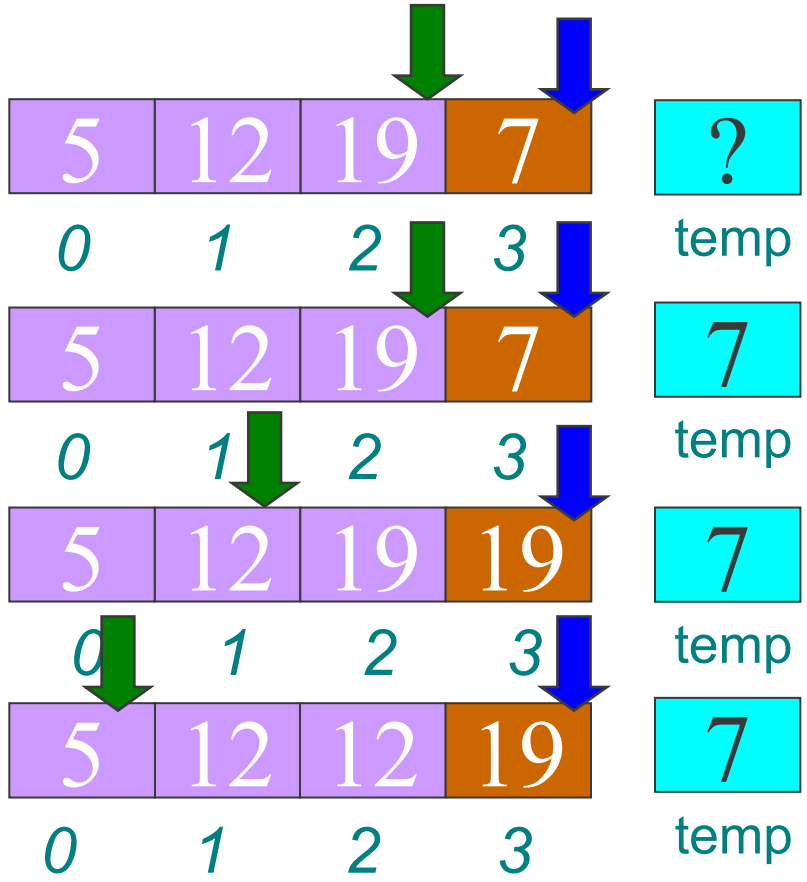
Insertion Sort – Example



- **Store unsorted in temp**
- **Shift bigger to right**
- **Store temp into freed**

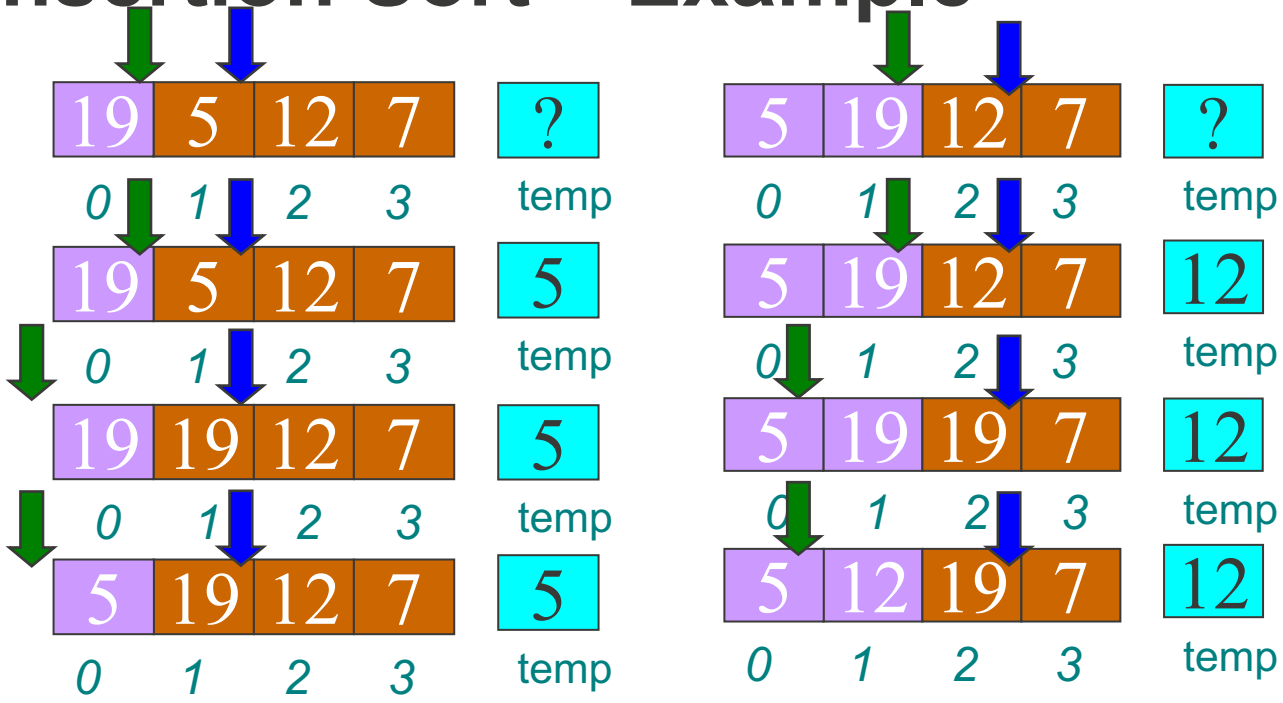
Insertion Sort – Example





- **Store unsorted in temp**
- **Shift bigger to right**
- **Store temp into freed**

Insertion Sort – Example

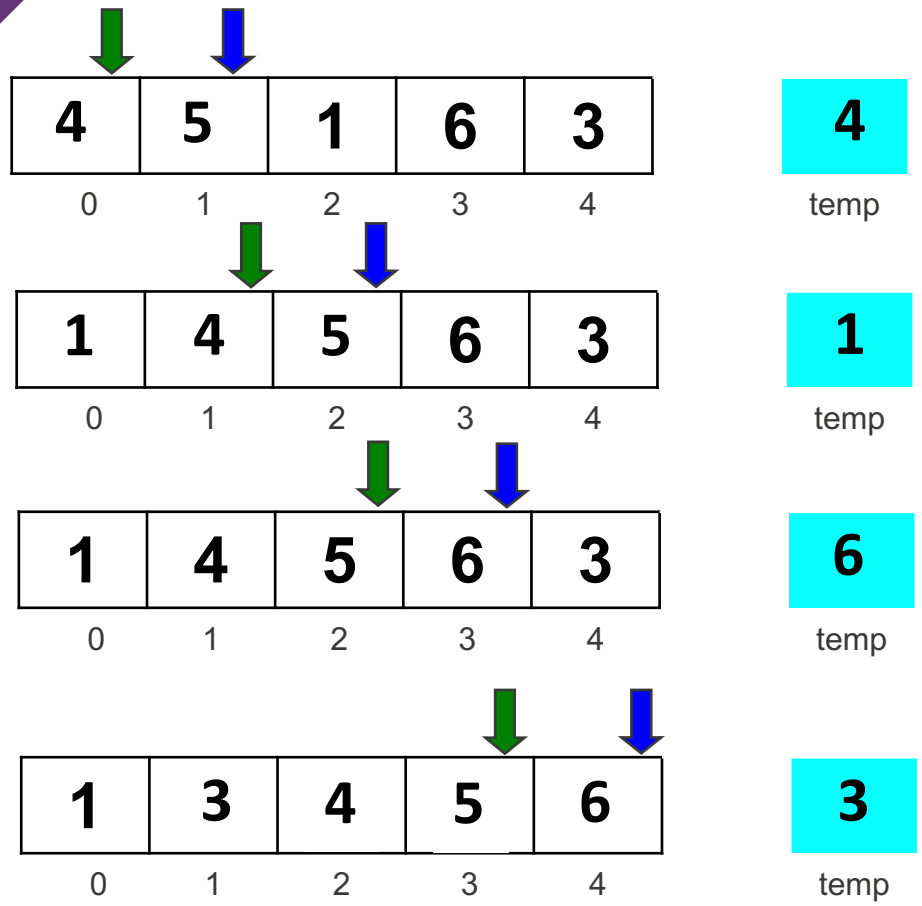


- **Store unsorted in temp**
- **Shift bigger to right**
- **Store temp into freed**

starts at 1, stops at n
starts at -1, stops at -1 or if element <= temp
temp moves into +1

Insertion Sort: Code

```
def insertion_sort(the_list):
    n = len(the_list)
    for mark in range(1,n):
        temp = the_list[mark]
        i = mark - 1
        while i >= 0 and the_list[i] > temp:
            the_list[i+1] = the_list[i]
            i -= 1
        the_list[i+1] = temp
```




```
def insertion_sort(the_list):
    n = len(the_list)
    for mark in range(1,n):
        temp = the_list[mark]
        i = mark - 1
        while i >= 0 and the_list[i] > temp:
           the_list[i+1] = the_list[i]
            i-= 1
        the_list[i+1] = temp
```

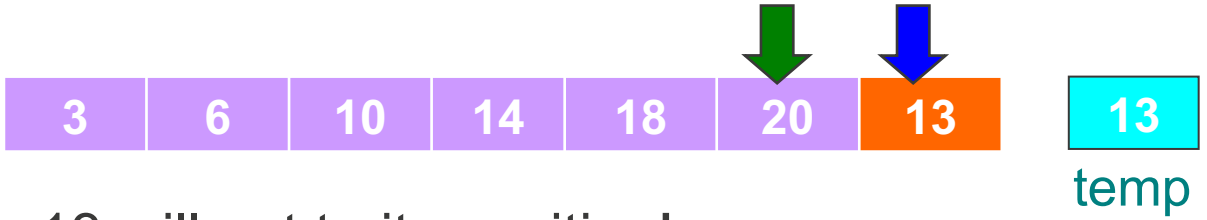
Properties of sorting algorithms

Algorithm	Stable	Incremental
Bubble Sort	Yes (strict)	No
Bubble Sort II	Yes (strict)	Yes (add to front)
Selection Sort	No	No
Insertion Sort		

Is this Insertion Sort incremental?

- **Consider again the sorted list**

- **If we now receive element 13, can Insertion Sort handle it incrementally?**
- **If we appended to the end AND put the mark at the new elements:**



  - In the first iteration 13 will get to its position!
- **How come we can now put the mark at the last sorted?**
  - Because of the invariant:
    - everything to the left is sorted but might not be in its final position
- **So yes, it is incremental by appending at the end and**

Is this Insertion Sort stable?

```
def insertion_sort(the_list):
    n = len(the_list)
    for mark in range(1,n):
        temp = the_list[mark]
        i = mark - 1
        while i >= 0 and the_list[i] > temp:
            the_list[i+1] = the_list[i]
            i -= 1
        the_list[i+1] = temp
```

8	3	8	6	3
a	b	c	d	e

The only time in which elements can get out of order is when we shuffle elements to the right, and when we re-insert the element from temp

The shuffled elements are kept in relative order, so that does not break stability

The temp element jumps over those shuffled, but none of the shuffled elements are = to temp, so that does not break stability either,

So, yes, it is stable

Can we ensure a and b are always before c and e, respectively?

Yes, but again, careful: a small change (>= rather than >) makes it non stable

Properties of sorting algorithms

Algorithm	Stable	Incremental
Bubble Sort	Yes (strict)	No
Bubble Sort II	Yes (strict)	Yes (add to front)
Selection Sort	No	No
Insertion Sort	Yes (strict)	Yes (add to back)

Summary

- **After these lesson you are now able to:**
  - Implement, use and modify the following sorting algorithms:
    - Bubble Sort
    - Selection Sort
    - Insertion Sort
  - Determine important invariants of sorting algorithms and use the invariants to improve them
  - In particular, you can reason about the stability and incrementality of sorting algorithms
- **We expect you to be able to do this for other simple algorithms that we may give you (not recursive ones yet)**