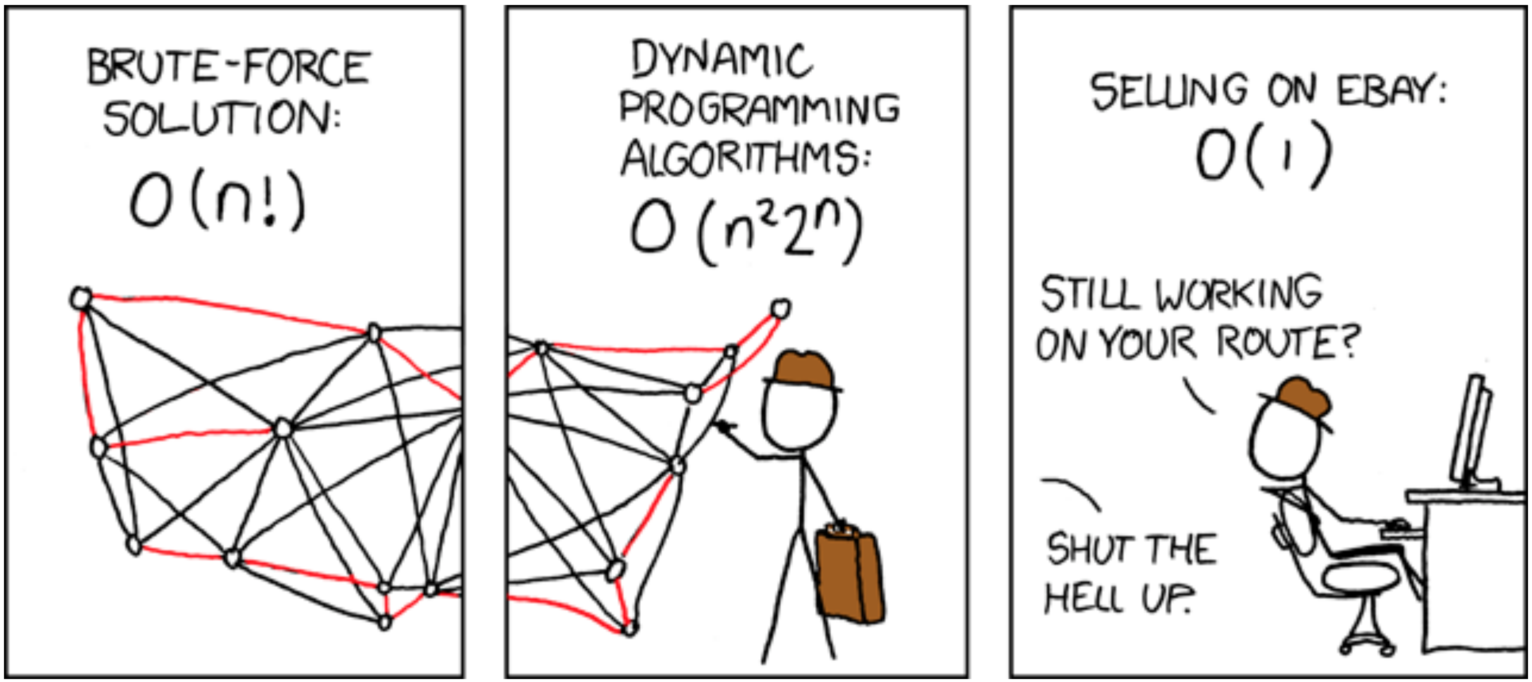


https://xkcd.com/399/

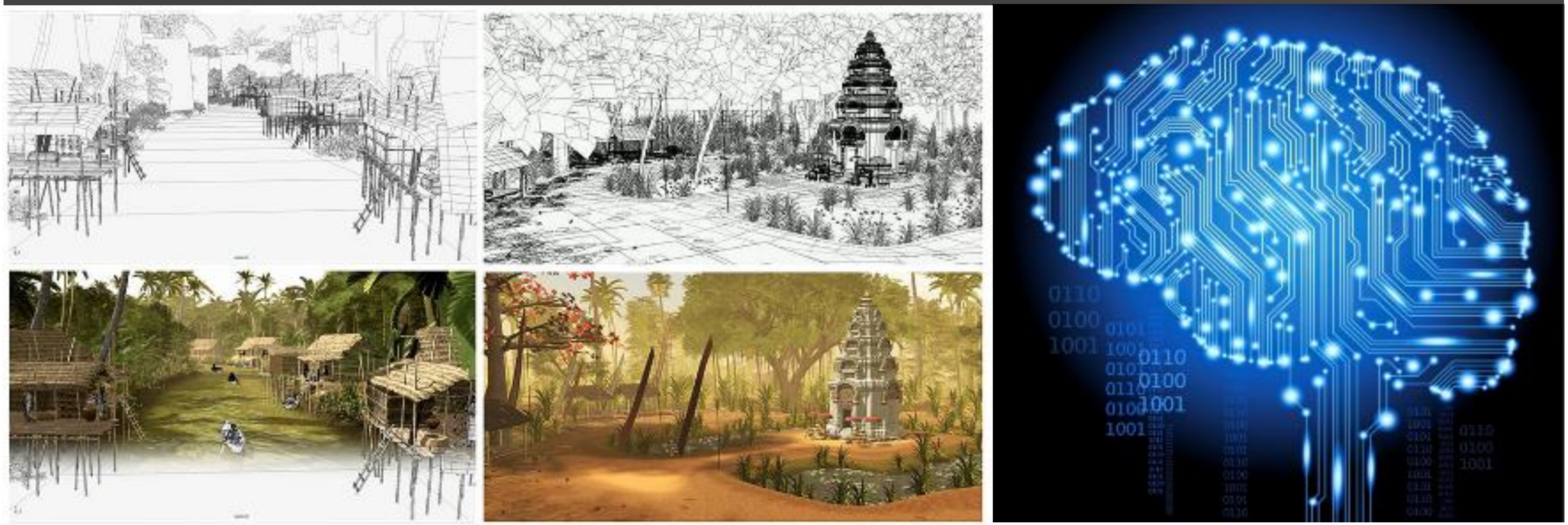
MONASH University

Information Technology

FIT2085 Lectures 10 and 11

Prepared by: M. Garcia de la Banda
based on D. Albrecht, J. Garcia

Sorting lists & Complexity



Where are we at?

- **During last three weeks we learned about MIPS**
- **We are now able to:**
  - Translate high-level code into MIPS with:
    - Simple arithmetic
    - Function call/return (even recursive functions)
    - If-then-elses and loops
    - Local and global variables
    - Integers and arrays
  - Discuss pros/cons MIPS architecture decisions
  - Reason about memory management in MIPS
  - Draw memory diagrams

Objectives for these two lectures

- **To understand the basic list sorting algorithms:**
  - Bubble Sort
  - Selection Sort
  - Insertion Sort
- **To be able to implement, use and modify them in Python**
- **To learn about running time and Big O time complexity**
  - To be able to compute the Big O complexity of simple functions
- **To reason about the properties of sorting algorithms:**
  - Invariants
  - Big O complexity
- **To be able to use the invariants to improve them**

Sorting lists

Sorting lists (increasing order)

Example:

[6,4,2,1,3,5] → [1,2,3,4,5,6]

Sorting Lists (increasing order)

This is our precondition

- **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

This is our postcondition

- **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

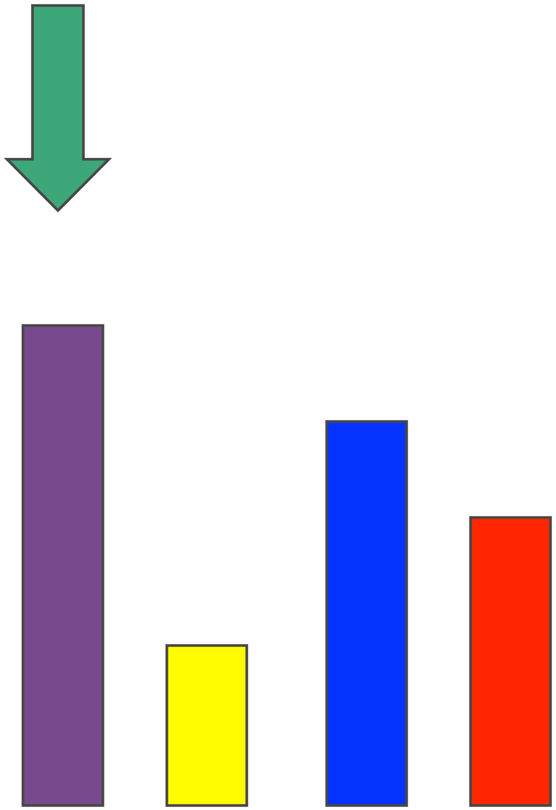
Bubble Sort

Bubble Sort

- **Very simple, so perfect for thinking about sorting**
- **Seen it in the 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

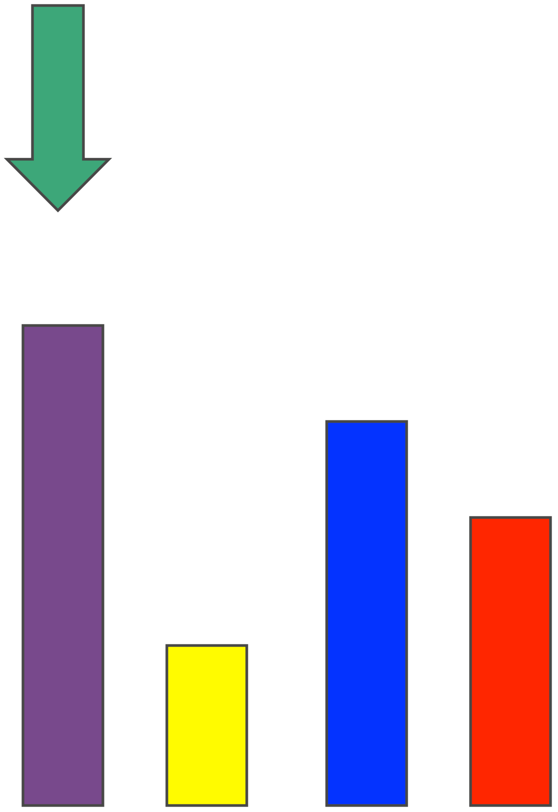
Bubble Sort Iteration: Example

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



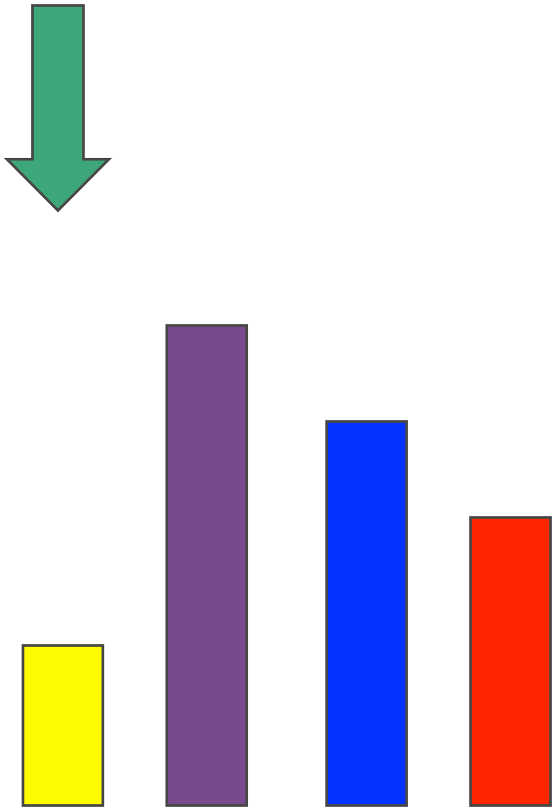
Bubble Sort Iteration: Example

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



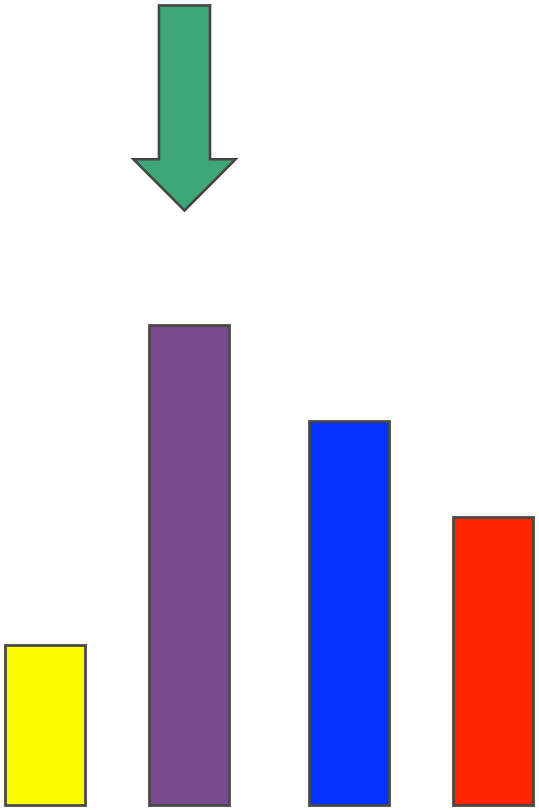
Bubble Sort Iteration: Example

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



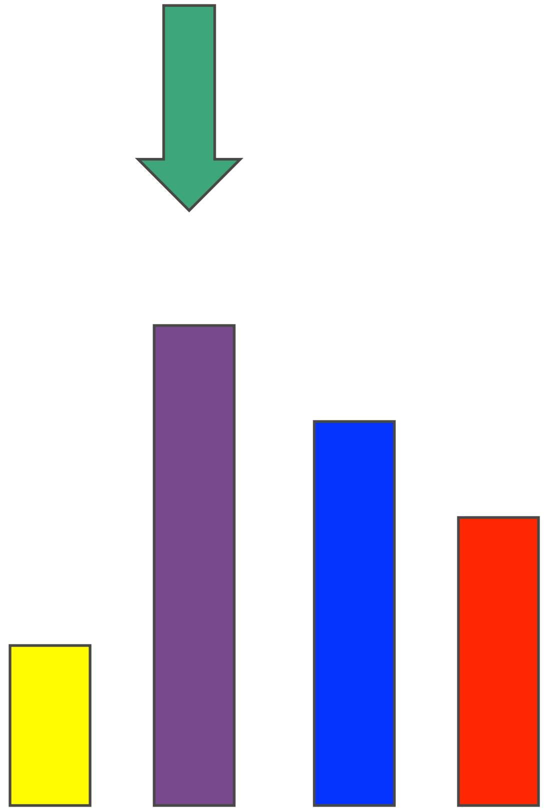
Bubble Sort Iteration: Example

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



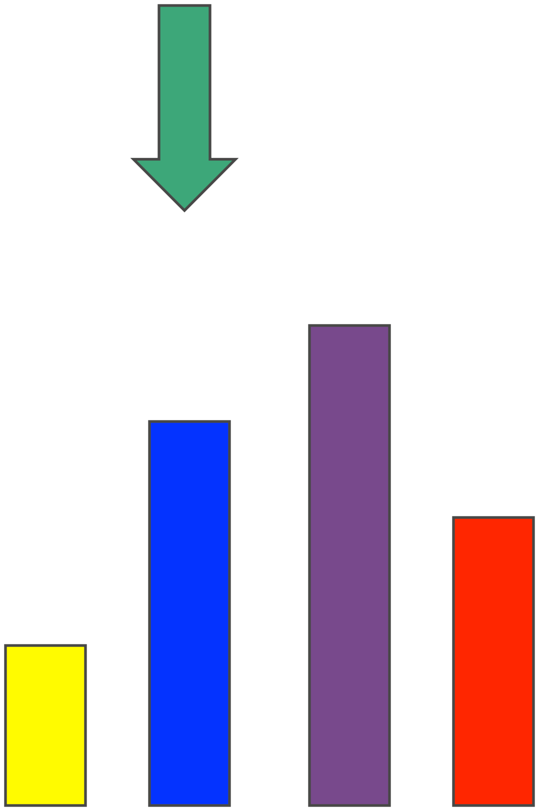
Bubble Sort Iteration: Example

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



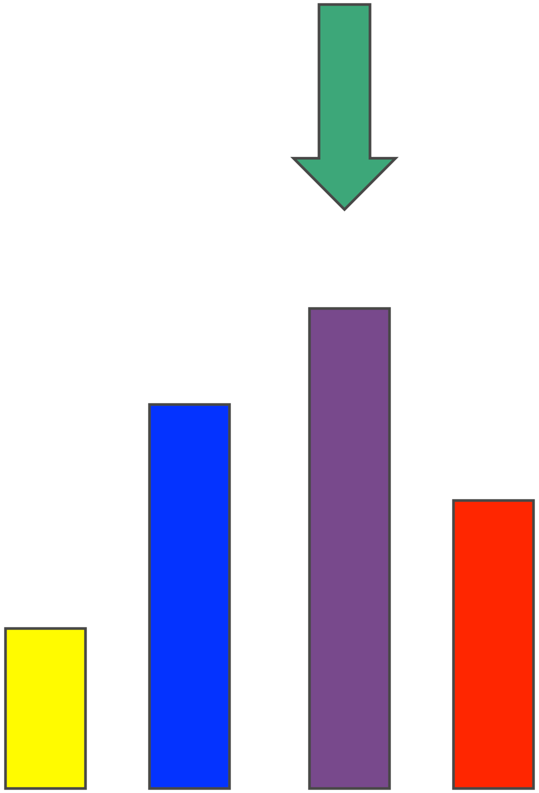
Bubble Sort Iteration: Example

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



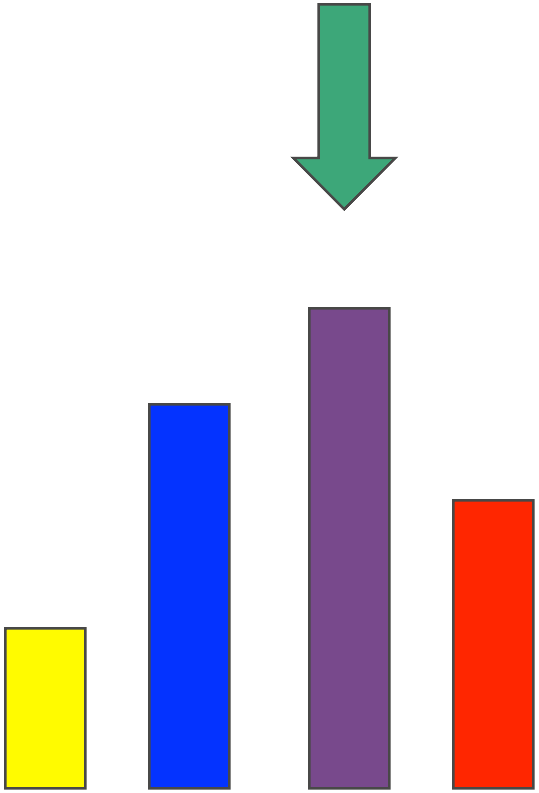
Bubble Sort Iteration: Example

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



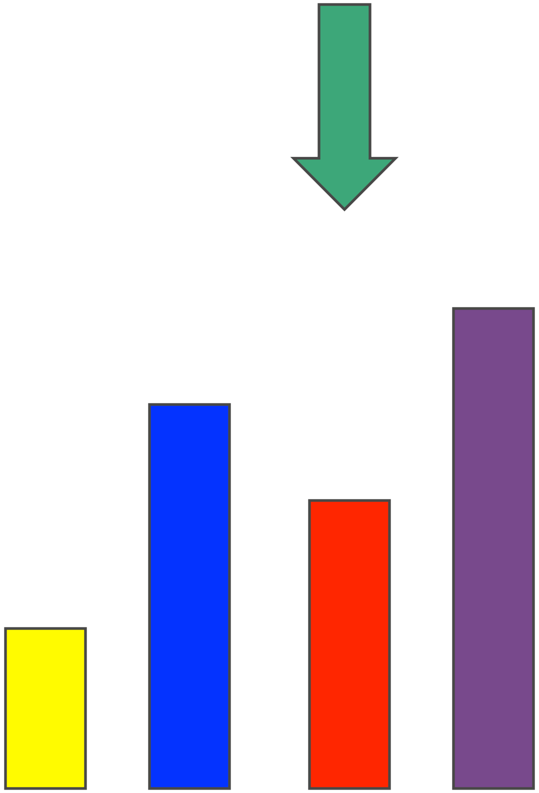
Bubble Sort Iteration: Example

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



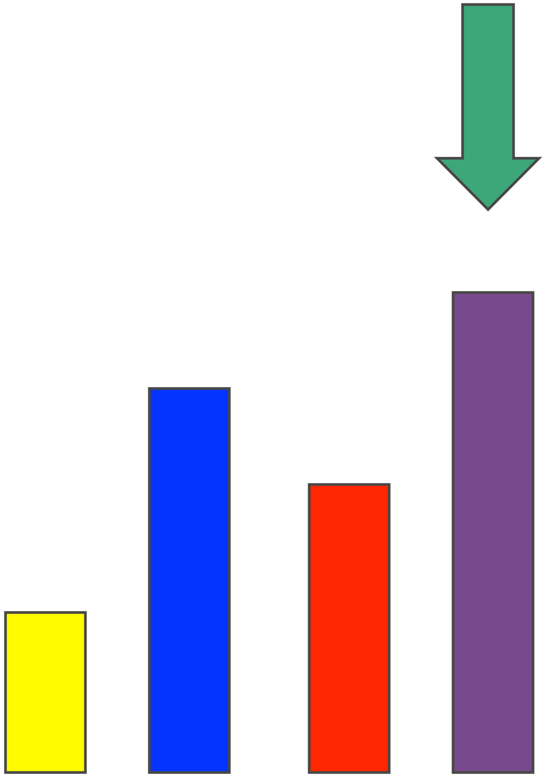
Bubble Sort Iteration: Example

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



Bubble Sort Iteration: Example

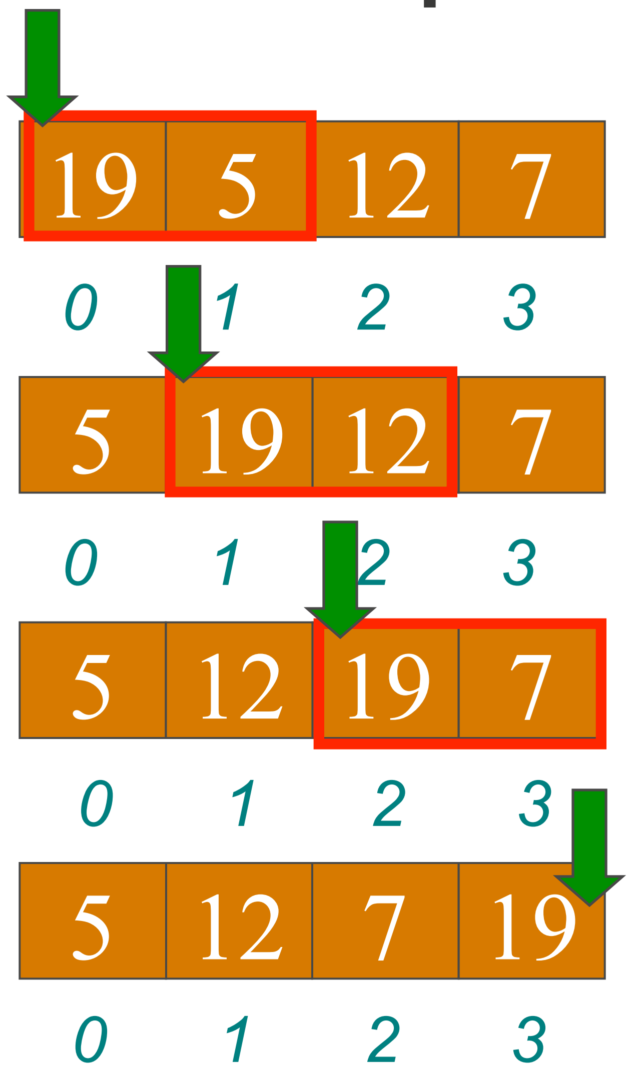
- Start at leftmost X
- If X > Y, swap them
- Move to right



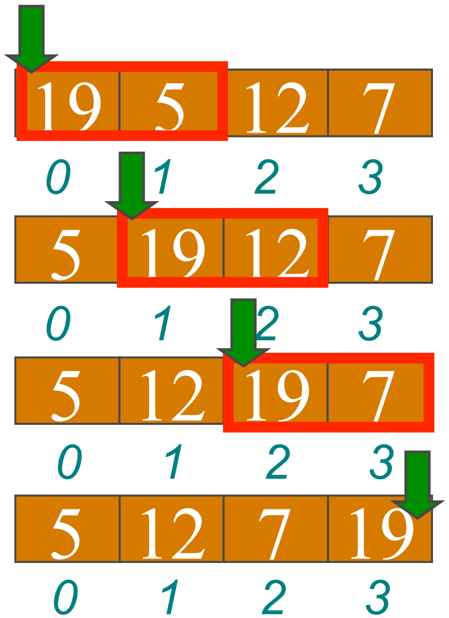
Bubble Sort: Invariants

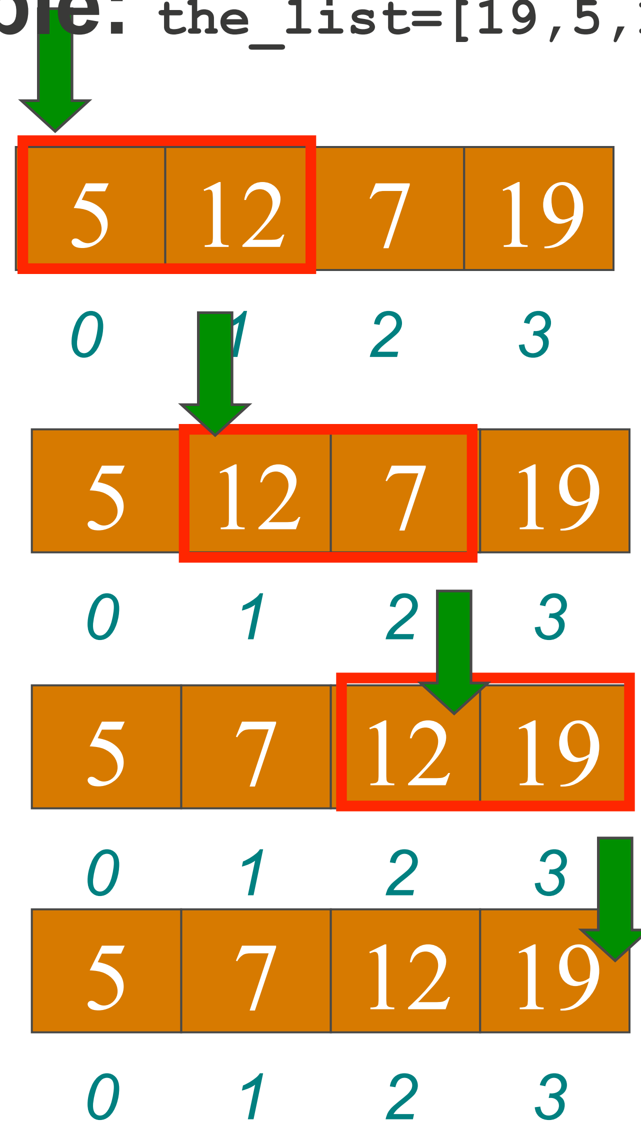
- **Invariant: property that remains unchanged**
  - At a particular point, or throughout an algorithm, or …
- **In bubble sort there are many invariants:**
  - Example: after every traversal, the list has the same elements
  - Also: in each traversal at most n-1 swaps are performed, where n is the length 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]`

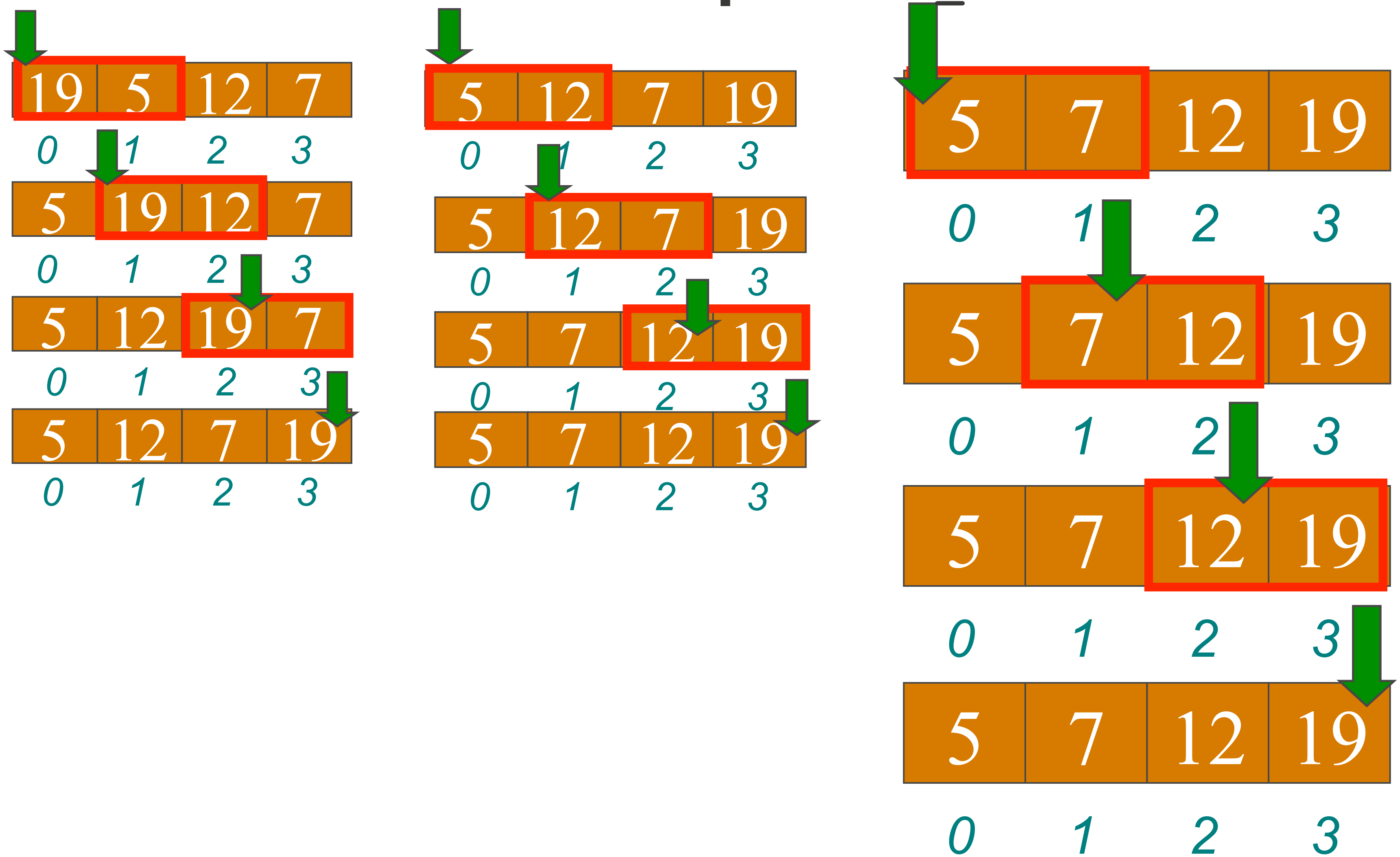


Bubble Sort – Example: `the_list=[19,5,12,7]`

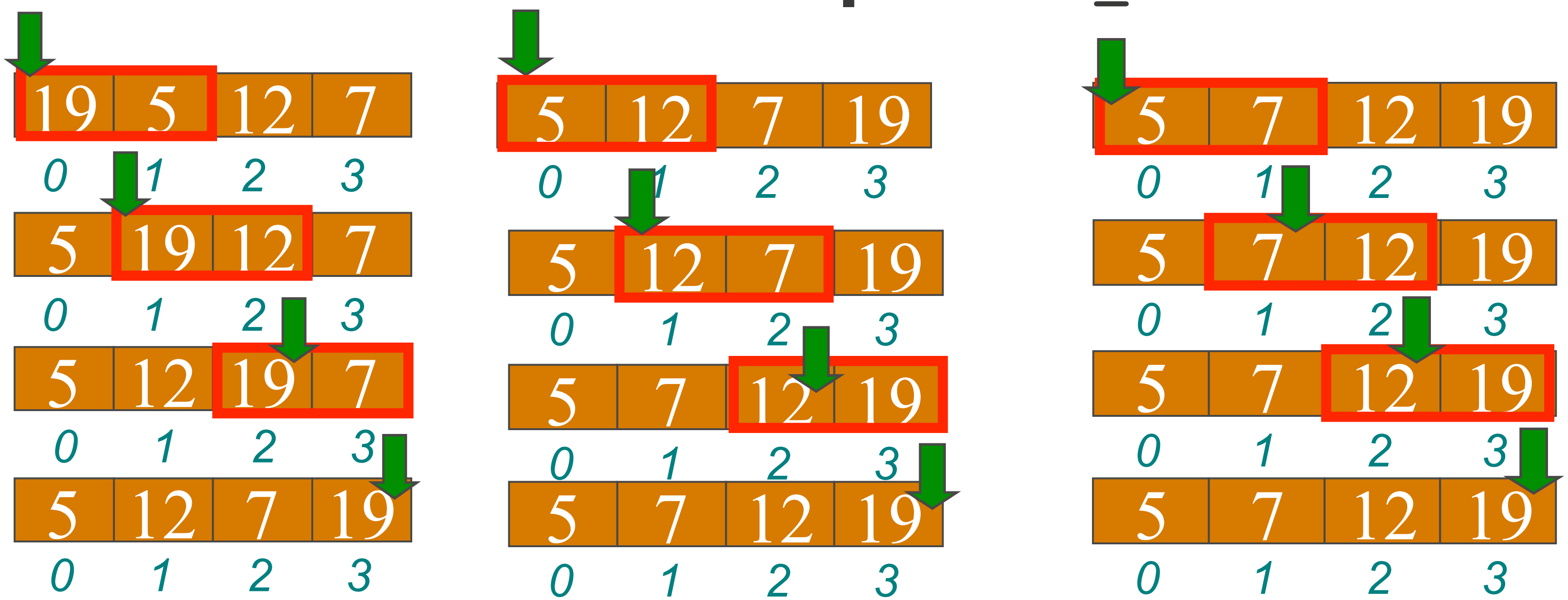




Bubble Sort – Example: `the_list=[19,5,12,7]`



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

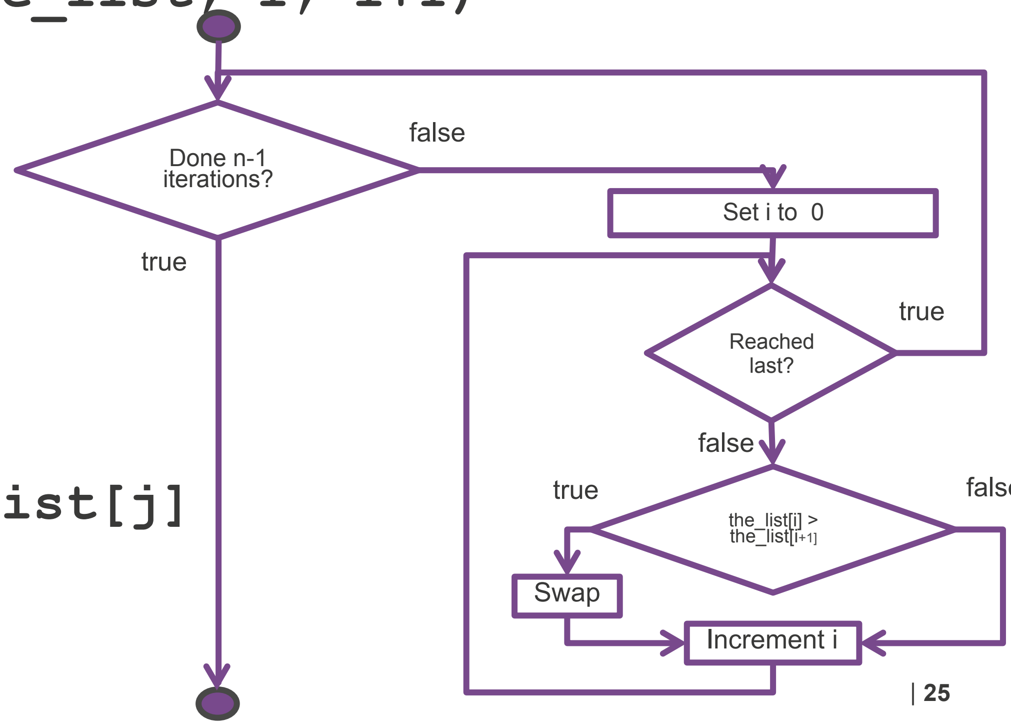


n-1 (3) passes performed

Bubble Sort: Python Code


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

`j` is never used. A way to iterate n-1 times. In Python you can use _ for unused variables




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

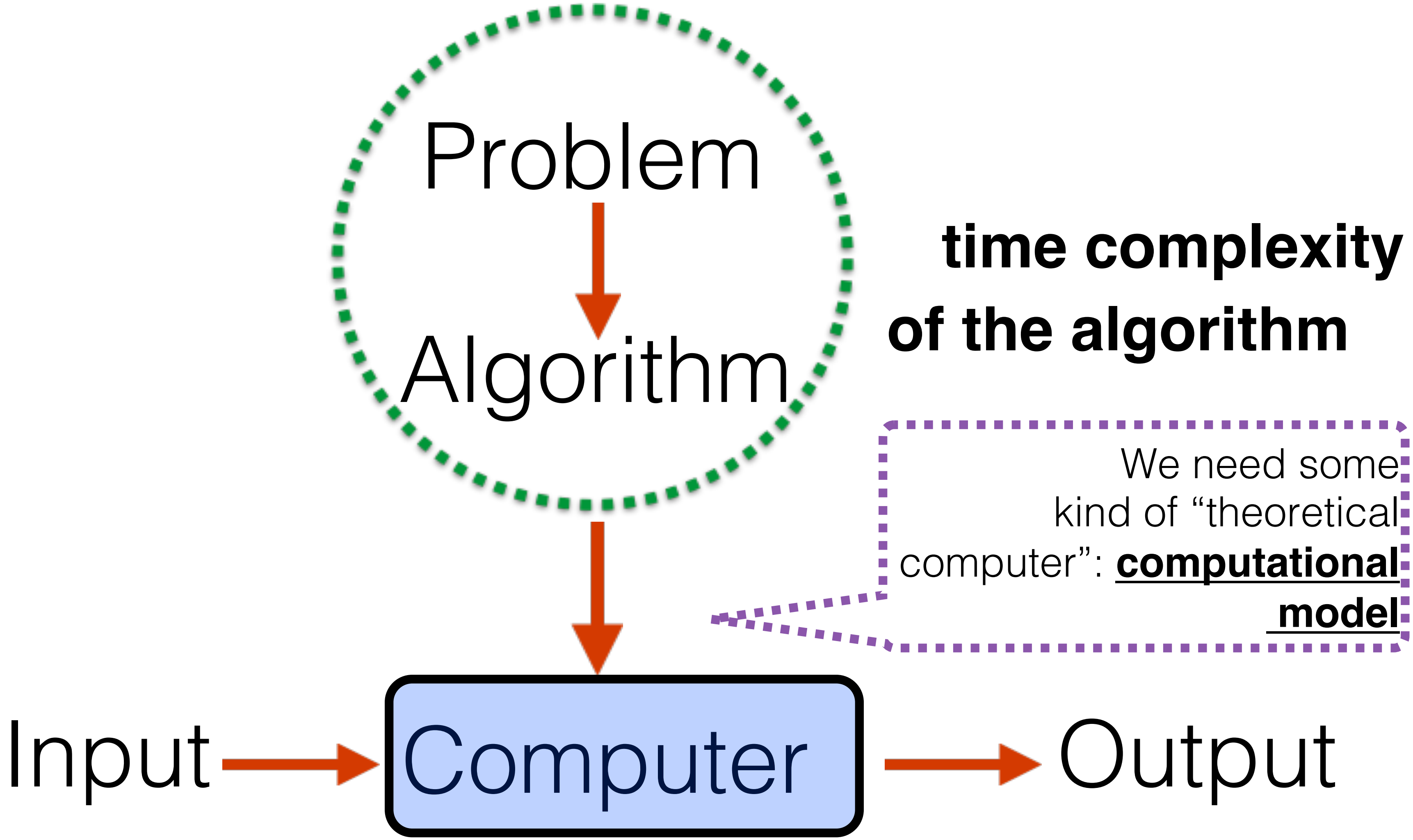
Running Time

Running time

Depends on a number of factors including:

- **The input**
- **The quality of the code generated by the compiler**
- **The nature and speed of the instructions on the machine executing the program**
- **The time complexity of the algorithm**

Jamaica's Usain Bolt celebrating after winning the final of the men's 100 metres athletics event at the 2015 IAAF World Championships in Beijing. AFP PHOTO / PEDRO UGARTE


Problem
Algorithm
time complexity of the algorithm
We need some kind of "theoretical computer": **computational model**
Input
Computer
Output

Simple computational model

- **Each simple statement/operation takes one “step”:**
  - Read, print and comparisons of numbers and booleans
  - Python list access
  - Assignments and basic numerical operations
  - Return statements
- **Sequence of statements: sum of their steps**
- **If-then-else: sum of the test plus branch(es)**
- **For a loop: sum of it statements times number of iterations**
  - Careful with nested loops (multiply inner and outer loop’s iterations)
- **Function calls: computed from its statements**

Example with Bubble Sort

```
def bubble_sort(the_list):
    n = len(the_list)          1 access and 1 assignment
    for _ in range(n-1):       2 assignments, 1 comparison, 1 increment
        for i in range(n-1):   2 assignments, 1 comparison, 1 incr
            if (the_list[i] > the_list[i+1]):   2 accesses, 1 comparison
                swap(the_list, i, i+1)   7 steps
```

n-1 times

n-1 times

The first assignment is outside the loop

1+1+1+(n-1)*(1+1+1+1+(n-1)*(1+1+1+2+1+7))=3+(n-1)*(4+(n-1)*13)) = 3+(n-1)*(13n-9) = 3+(13n^2-22n+9) = 13n^2-22n+12 “steps” for bubble sort ….

Wow!!! Is all this detail accurate? Useful?

```
def swap(the_list,i,j):
    tmp = the_list[i]              1 access and 1 assignment
    the_list[i]=the_list[j]        2 accesses and 1 assignment
    the_list[j] = tmp              1 access and 1 assignment
```

1+1+2+1+1+1=7 “steps” for swap

Big O time complexity

Big O notation for time complexity

- **The exact running time function T(n), where n is the size of the input data, can be difficult to compute and understand**
- **Instead: compute an upper bound f(n) to T(n) that**
  - Ignores parts of **T(n)** that do not add significantly to the total running time
  - Bounds the error made when ignoring these small parts
- **Gives us a way of describing the growth rate of a method**
  - Behaviour when its input arguments grow towards infinity

- **Formally: function T(n) is said to be O(f(n)) if there exist constants k and L such that**

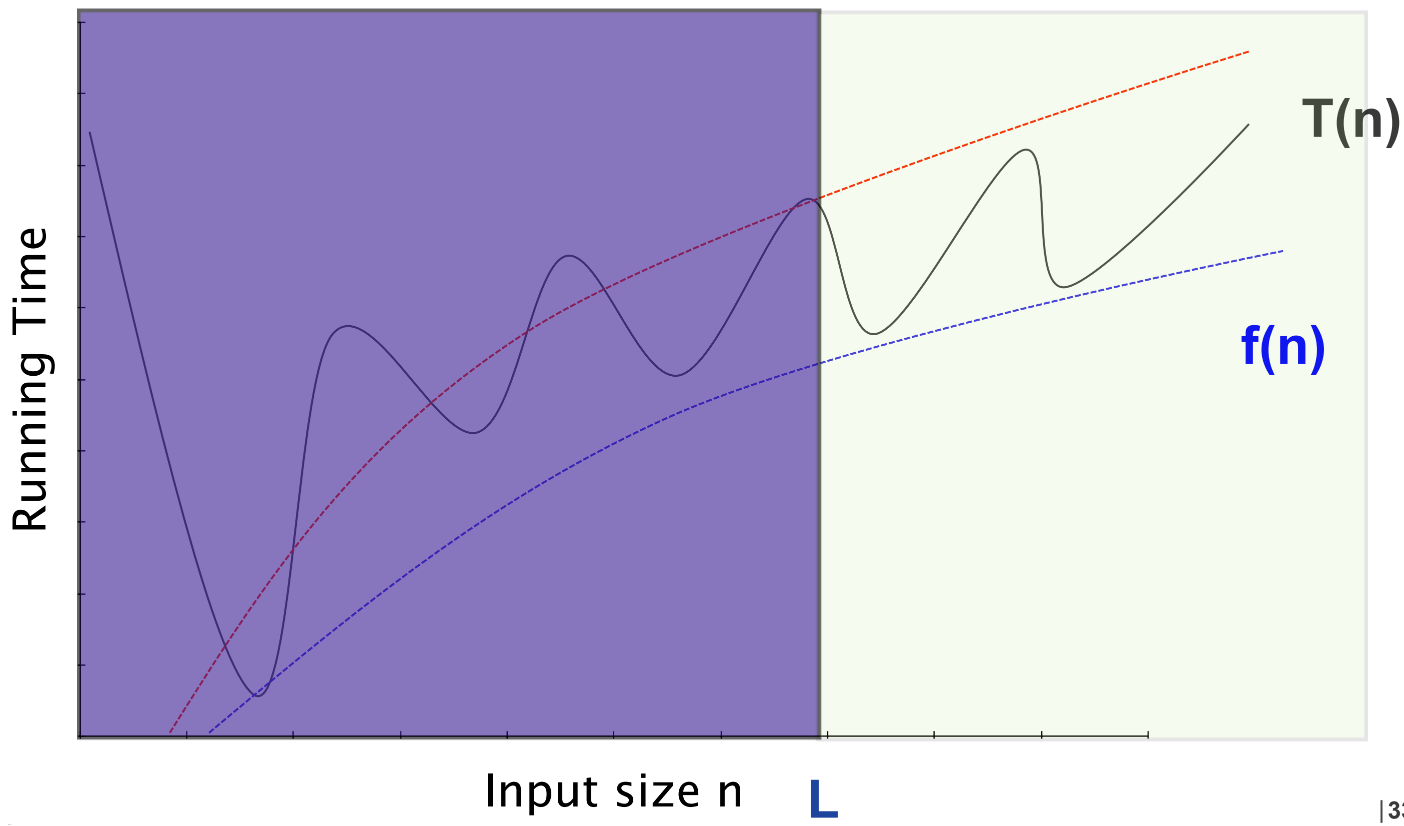
$$T(n) \leq k * f(n) \text{ for all } n > L$$

Big O notation for time complexity

- Function T(n) is said to be O(f(n)) if there exist constants k and L such that

T(n) ≤ k*f(n) for all n > L

Big O gives us an idea of T(n)’s growth behaviour for large inputs. Simple but formal.



Common Big O efficiency classes

Constant	O(1)	Running time does not depend on N	N doubles, T remains constant
Logarithmic	O(log N)	Problem is broken up into smaller problems and solved independently. Each step cuts the size by a constant factor.	If N doubles, running time T gets slightly slower
Linear	O(N)	Each element requires a certain (fixed) amount of processing	If N doubles, running time T doubles (2*T)
Superlinear	O(N log N)	Problem is broken up in sub-problems. Each step cuts the size by a constant factor and the final solution is obtained by combining the solutions.	If N doubles, running time T gets slightly bigger than double (2*T and a bit)
Quadratic	$O(N^2)$	Processes pairs of data items. Often occurs when you have double nested loop	If N doubles, running time T increases four times (4*T)
Exponential	$O(2^N)$	Combinatorial explosion (think about a family tree)	If N doubles, running time T squares (T*T)
Factorial	O(N!)	Finding all the permutations of N items	

Growth rates

N	log(N)	N	Nlog(N)	N^2	2^N	N!
10	0.003 µs	0.01 µs	0.033 µs	0.1 µs	1 µs	3.63 ms
20	0.004 µs	0.02 µs	0.086 µs	0.4 µs	1 ms	77.1 years
30	0.005 µs	0.03 µs	0.147 µs	0.9 µs	1 sec	8.4×10^{15} years
40	0.005 µs	0.04 µs	0.213 µs	1.6 µs	18.3 min	
50	0.006 µs	0.05 µs	0.282 µs	2.5 µs	13 days	
100	0.007 µs	0.1 µs	0.644 µs	10 µs	4×10^{13} years	
1,000	0.010 µs	1 µs	9.966 µs	1 ms		
10,000	0.013 µs	10 µs	130 µs	100 ms		
100,000	0.017 µs	100 µs	1.67 ms	10 sec		
1,000,000	0.020 µs	1 ms	19.93 ms	16.7 min		
10,000,000	0.023 µs	10 ms	0.23 sec	1.16 days		
100,000,000	0.027 µs	0.1 sec	2.66 sec	115.7 days		
1,000,000,000	0.030 µs	1 sec	29.90 sec	31.7 years		

Measured in nanoseconds (10^{-9} secs)

Main things to remember

- **Ignore constants**
  - It is not O(10n), just O(n)
- **Ignore parts that do not contribute significantly**
  - It is not $O(n^3 + n^2 + n)$, just $O(n^3)$
- **Always assume an unknown input size n for each argument**
  - n will be large (measuring growth towards infinity)
- **Makes things much easier!**
  - Don't need to worry about the exact number of steps
- **Why can you do this?**
  - It is an upper bound

Best/worst/average case complexity

- **Running time can depend on things OTHER than size**
  - Like the list being sorted, or having the element we look for first
- **Worst case: gives a guarantee (correct for all inputs)**
  - Most often-quoted
- **Best case: correct for at least one input**
  - Less useful. “Lucky” inputs may be rare
- **Average: describes “usual” behaviour, not extremes …**
  - Often tricky to work out, so not discussed in FIT2085
- **If run time depends only on input size: best = worst**
- **Together, worst & best give an idea of the range of possibilities**
- **In unspecified, “time complexity” means worst case**

Back to Bubble Sort

Back to Bubble Sort

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

n-1 times

n-1 times

So what is the complexity?

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

Constant run time for swap, so O(1)

Back to Bubble Sort: details

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

- **The inner loop**
  - Runs (n-1)*(n-1) times
  - The comparison is always performed
    - For now we will assume constant time comparison
    - This is often NOT true (e.g., if the elements are strings)
  - The swap might be performed
    - Always (list in reverse order)
    - Never (list already sorted)
    - Sometimes (common case)
  - Does not affect the number of iterations, only the constant:
    - Smallest if already sorted
    - Biggest if reversed

Bubble Sort: Time complexity

- **Approximating inner loop ops by a constant (k), we get:**

$$(n-1)*(n-1)*k = (n^2-2n+1)*k \rightarrow O(n^2)$$

- **That is the worst time complexity:**
  - Both loops run for the maximum number of iterations
- **What is the best time complexity?**
  - Any properties of the elements that reduce big O?
  - In this case: that stop any of the two loops early?
  - Being empty is NOT a property of the elements!
    - We are considering the scalability of the algorithm
    - So we must always assume a big n
- **No such property for the algorithm. This tells you what?**
  - best = worst

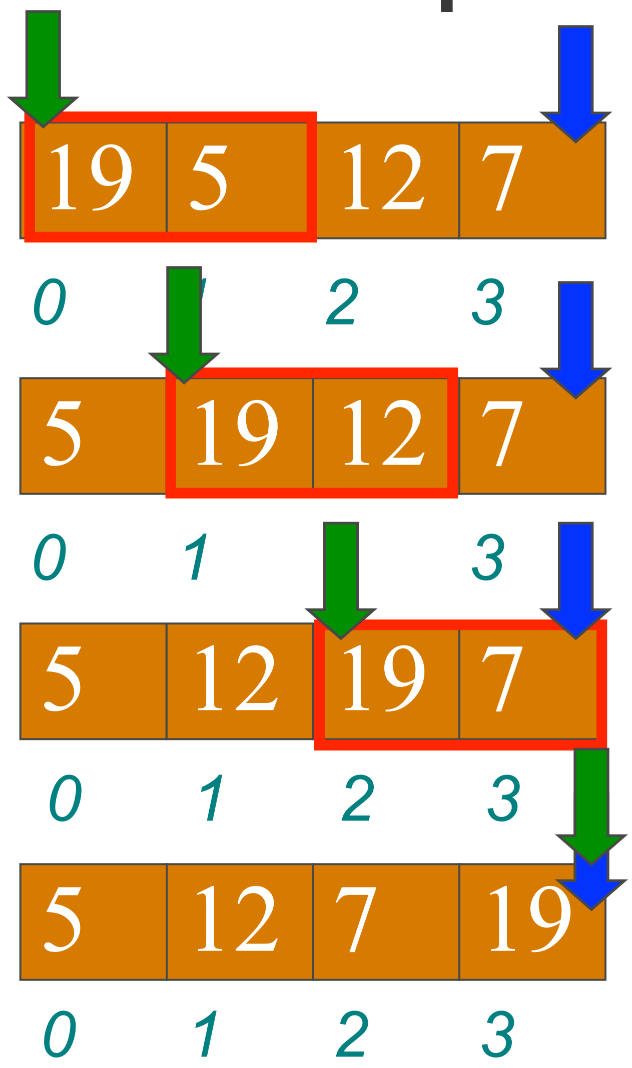
Properties of sorting algorithms

Algorithm	Best case	Worst case	Stable	Incremental
Bubble Sort	$O(n^2)$	$O(n^2)$		

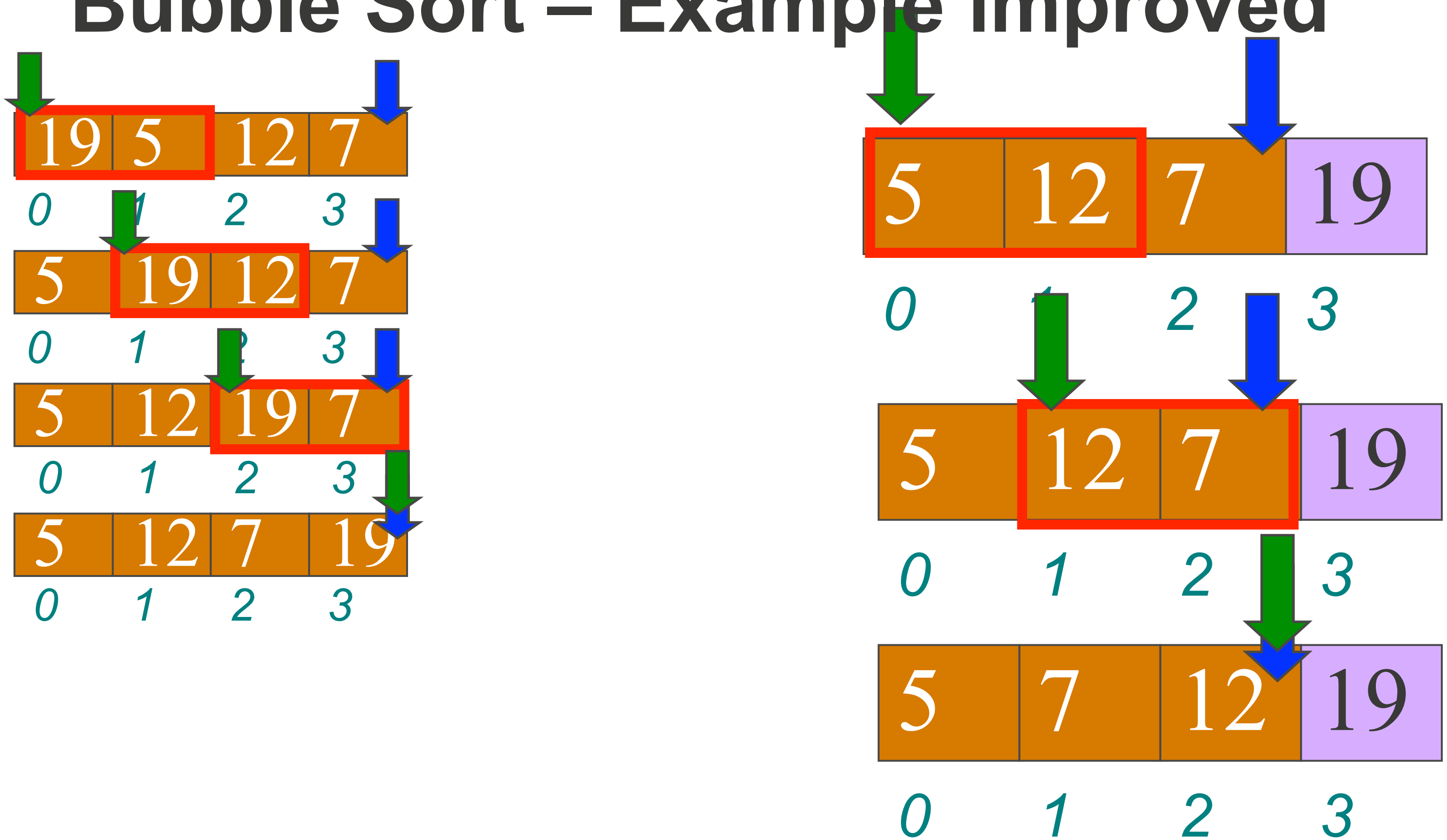
Bubble Sort: Optimization

- **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

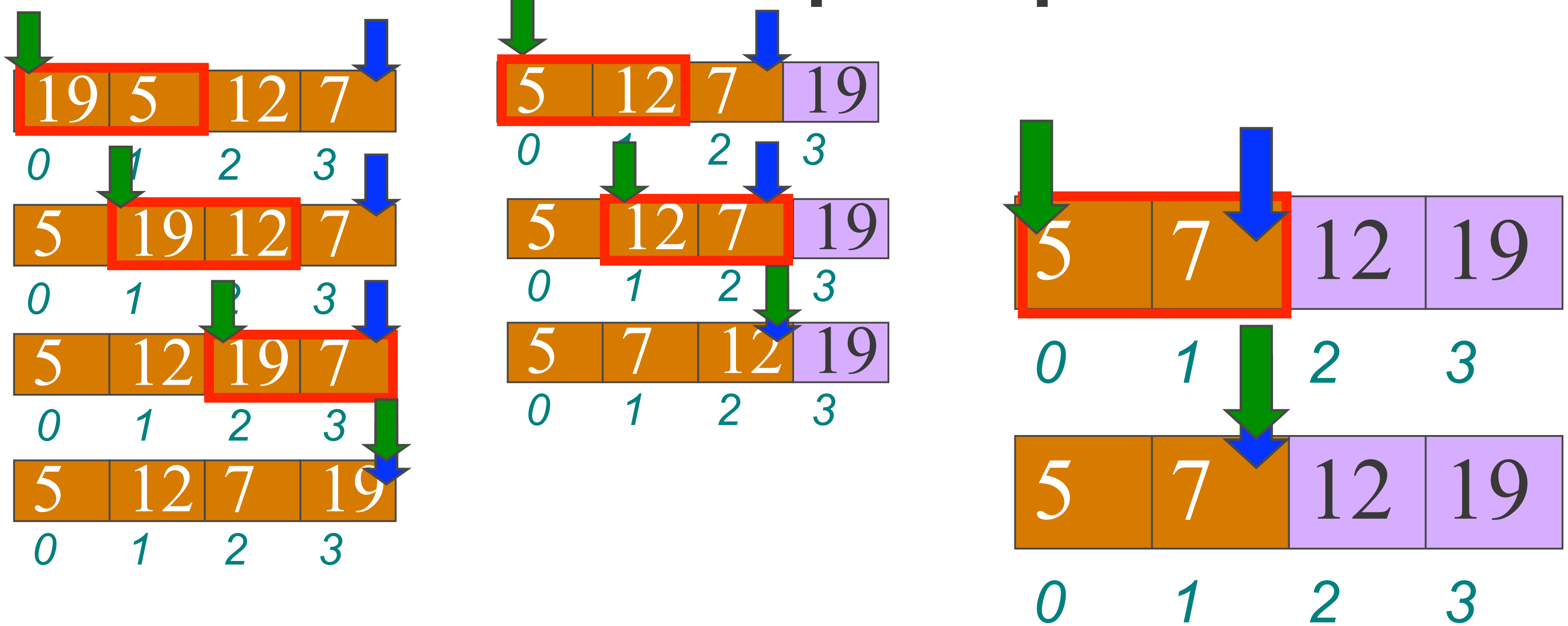
Bubble Sort – Example improved



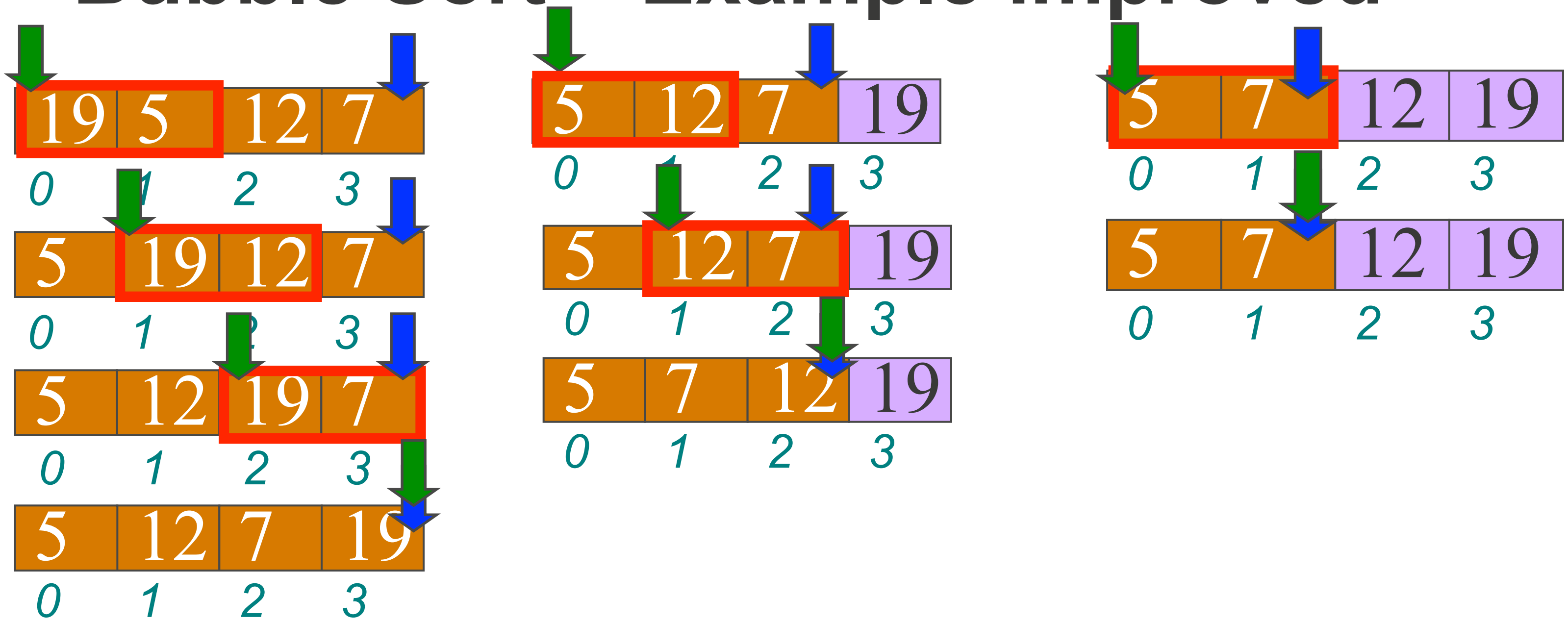
Bubble Sort – Example improved



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

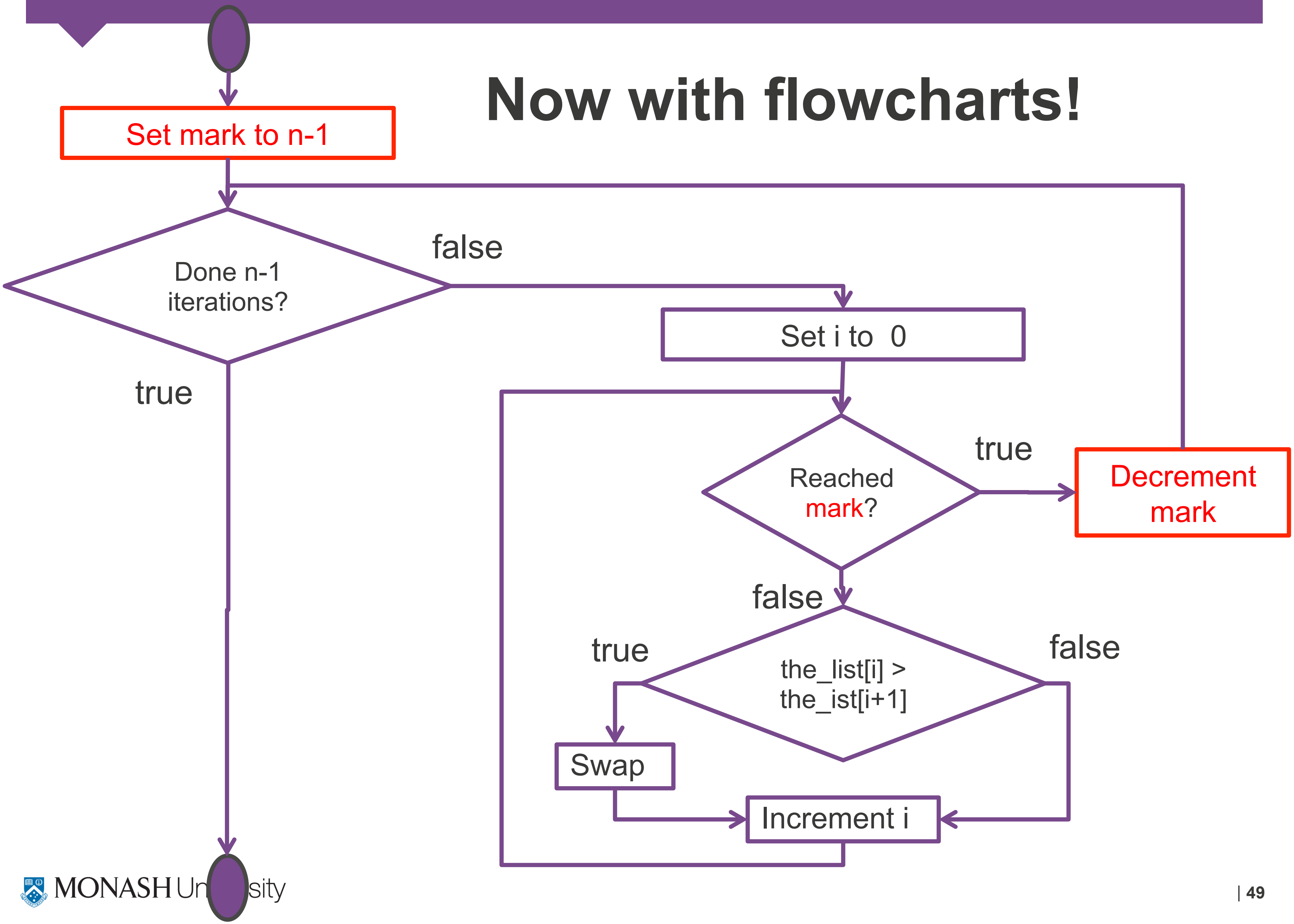
Bubble Sort: one possible algorithm

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

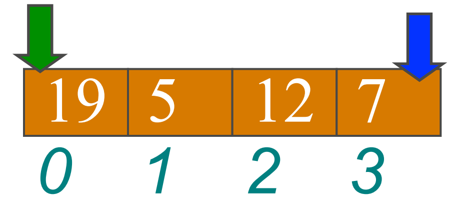
- **The mark needs to go from n-1 to 1 (stop at 0)**
- **How do we do this in Python?**
- **We have seen `range(n)`**
  - Goes from 0 incrementing by 1 until it reaches n
- **Also `range(start,stop[,step])`**
  - Goes from `start` incre/decrementing to by `step`, until it reaches `stop`
  - If `step` is omitted, its value is 1
  - If `start` is omitted, its value is 0

`list()` transforms the sequence into a list

```
>>> 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)
```

```python
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)
```

5	**4**	**1**	**6**	**3**
0	1	2	3	4

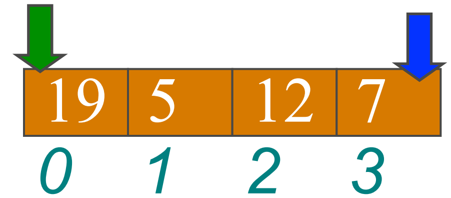
0	1	2	3	4

0	1	2	3	4

0	1	2	3	4

0	1	2	3	4

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)
```

n-1 times

mark times

constant

constant

Intuition: nested loops, both dependent on n, every operation on inner loop performed a fixed number of times: the worst case is going to be $O(n^2)$

Bubble Sort: Time complexity

- **The inner loop runs for:**

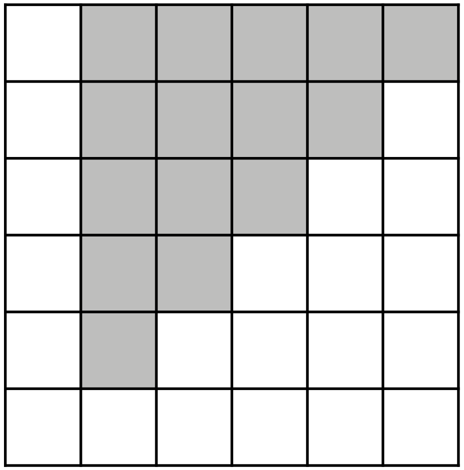
$$(n-1) + (n-2) + (n-3) + \ldots + 1 = n*(n-1)/2 = (n^2 - n)/2$$

  – The rest is as before

- **Approximating by a constant (say k), we have:**

$$k*(n^2-n)/2 \rightarrow O(n^2)$$

- **Any properties of the list elements that affect big O?**
  – In this case: that stop any of the two loops early?



- **No! This tells you what?**
  – best = worst
- **Same complexity than the previous version!**
- **So why is it better?**
  – Not better scalability BUT
  – Better efficiency (half the number of inner iterations)

Bubble Sort: More optimizations

- **Consider the list of elements:**

7	5	23	12	14	56	32	40	45

- **What happens after 1 iteration?**

5	7	12	14	23	32	40	45	56

- **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
- **How does this affect complexity?**

Bubble Sort: Python Code

19	5	12	7
0	1	2	3

```
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
```

? times

mark times

Not the best code, but easiest to show differences. A while outer loop with a condition on swapped would be better.

Breaks out the closest enclosing **`for`** or **`while`** loop

Does this change BigO complexity?

Best case is now O(n) when the list is sorted

Properties of sorting algorithms

Algorithm	Best case	Worst case	Stable	Incremental
Bubble Sort	$O(n^2)$	$O(n^2)$		
Bubble Sort II	$O(n)$	$O(n^2)$		

Is this 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**

3	6	10	14	18	20

- **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?**
  - 1^{st}

3	6	10	14	18	13	20

  - 2^{nd}

3	6	10	14	13	18	20

  - 3^{rd}

3	6	10	13	14	18	20

  - 4^{th}: detect no swaps and finish

- **Not very incremental**
  - If the new element is the smallest: runs all iterations

Is this algorithm incremental? (cont)

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

13	3	6	10	14	18	20

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

3	6	10	13	14	18	20

  - 2^{nd}: detect no swaps and finish
- **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, need to shuffle everything to the right
  - And this already takes one iteration
  - Appending to the end is constant time (rather than O(n))

Properties of sorting algorithms

Algorithm	Best case	Worst case	Stable	Incremental
Bubble Sort	$O(n^2)$	$O(n^2)$		Yes (add to front)
Bubble Sort II	$O(n)$	$O(n^2)$		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?

```
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

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 a small change (>= rather than >) makes it non stable

Properties of sorting algorithms

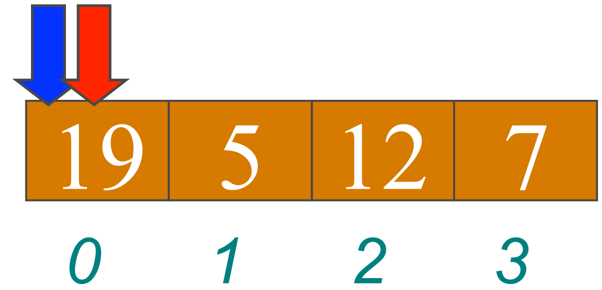
Algorithm	Best case	Worst case	Stable	Incremental
Bubble Sort	$O(n^2)$	$O(n^2)$	Yes (strict)	Yes (add to front)
Bubble Sort II	$O(n)$	$O(n^2)$	Yes (strict)	Yes (add to front)

Selection Sort

Selection Sort

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

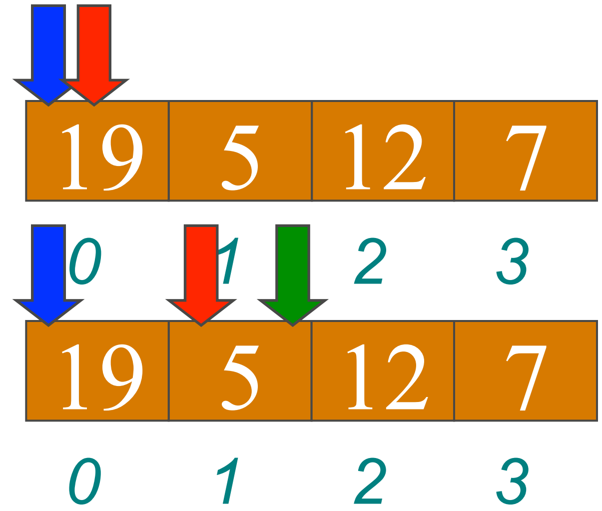
Selection Sort – Example



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

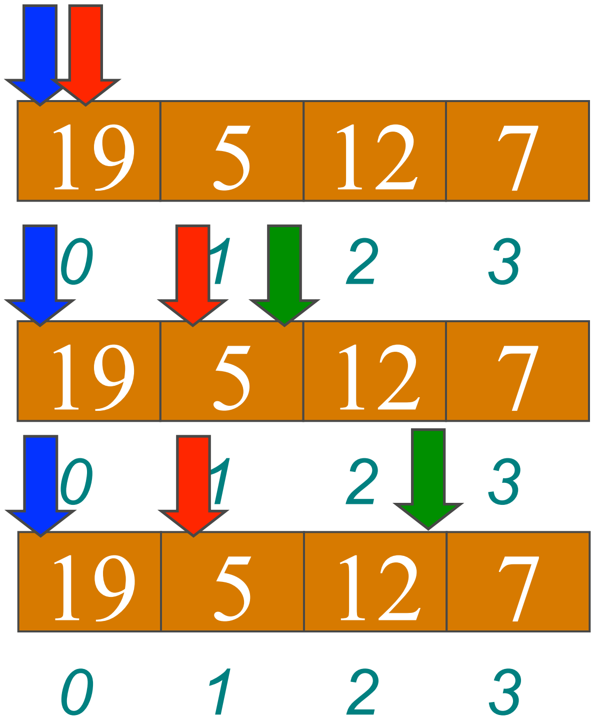
Selection Sort – Example

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



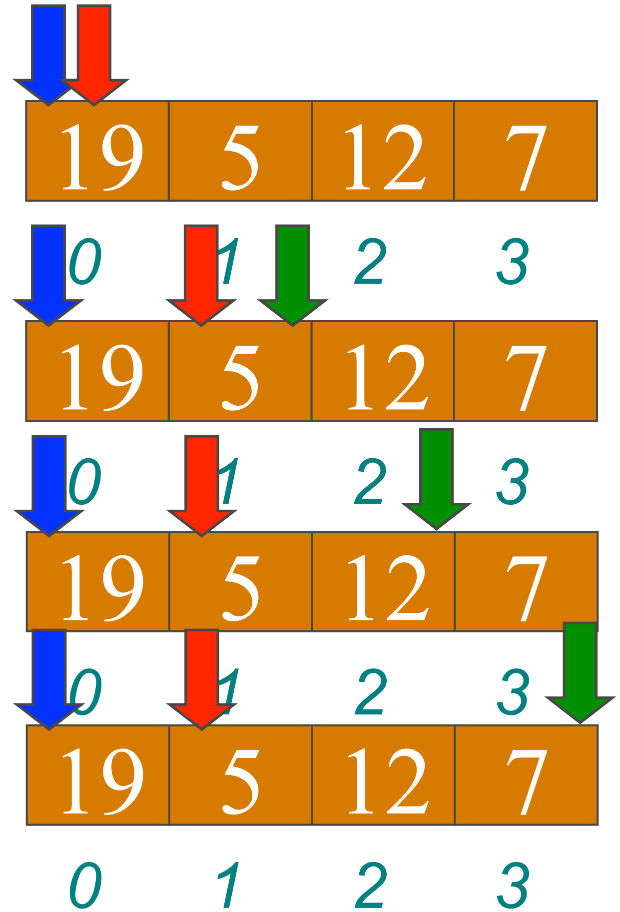
Selection Sort – Example

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



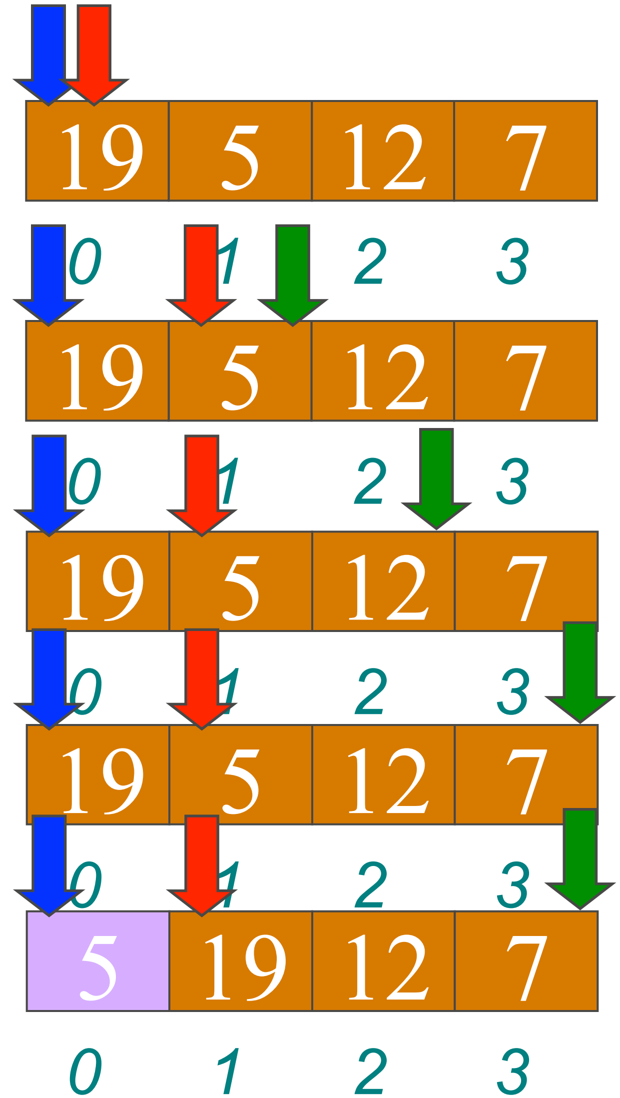
Selection Sort – Example

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

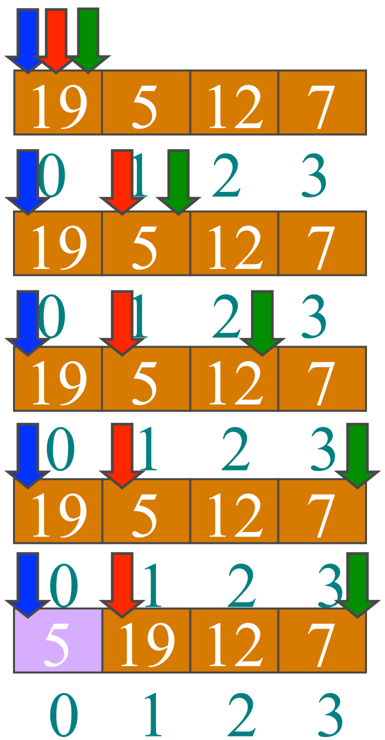


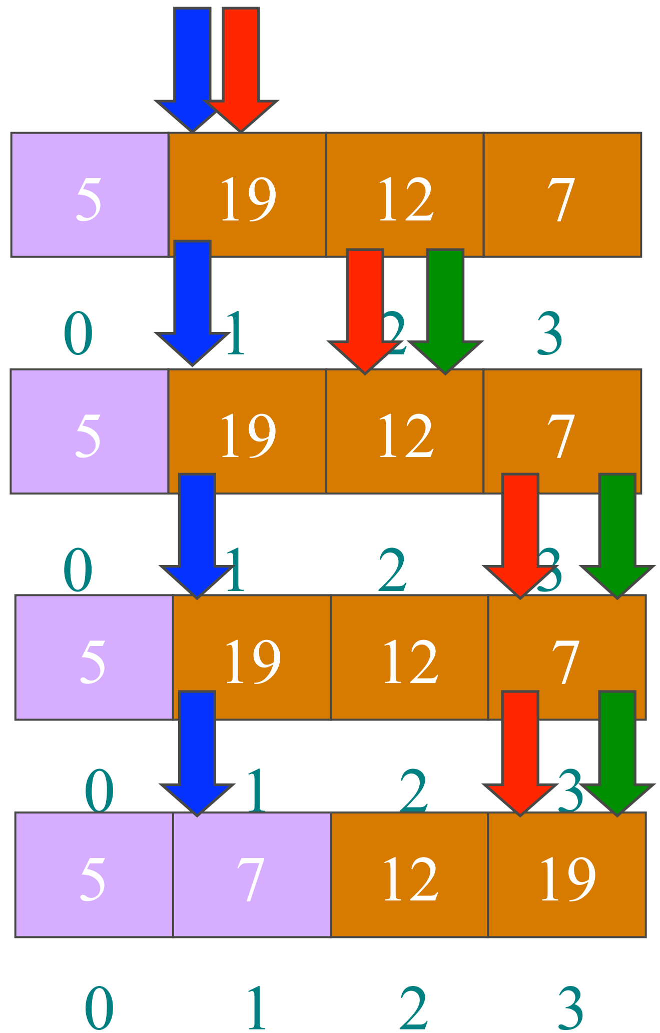
Selection Sort – Example

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

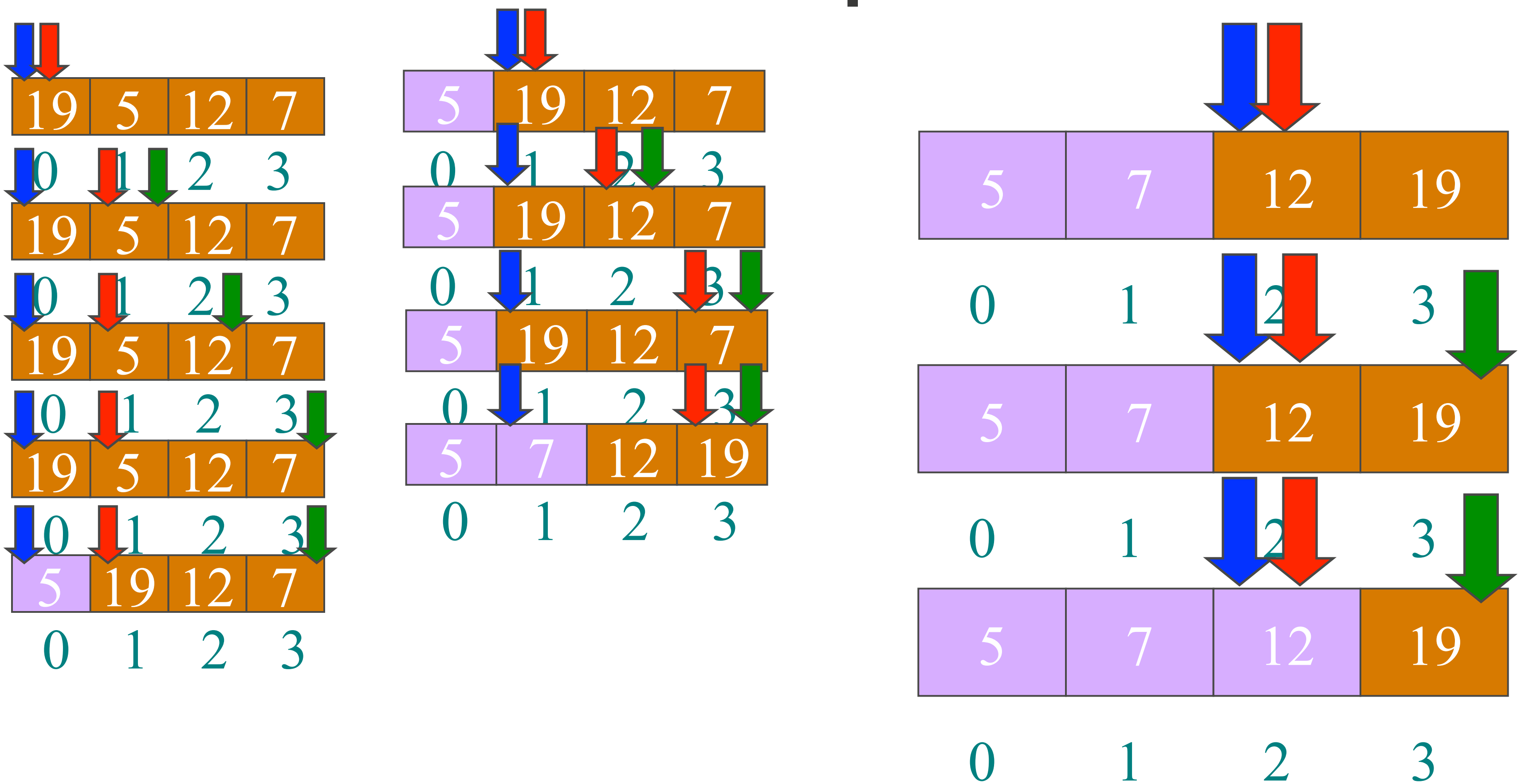


Selection Sort – Example

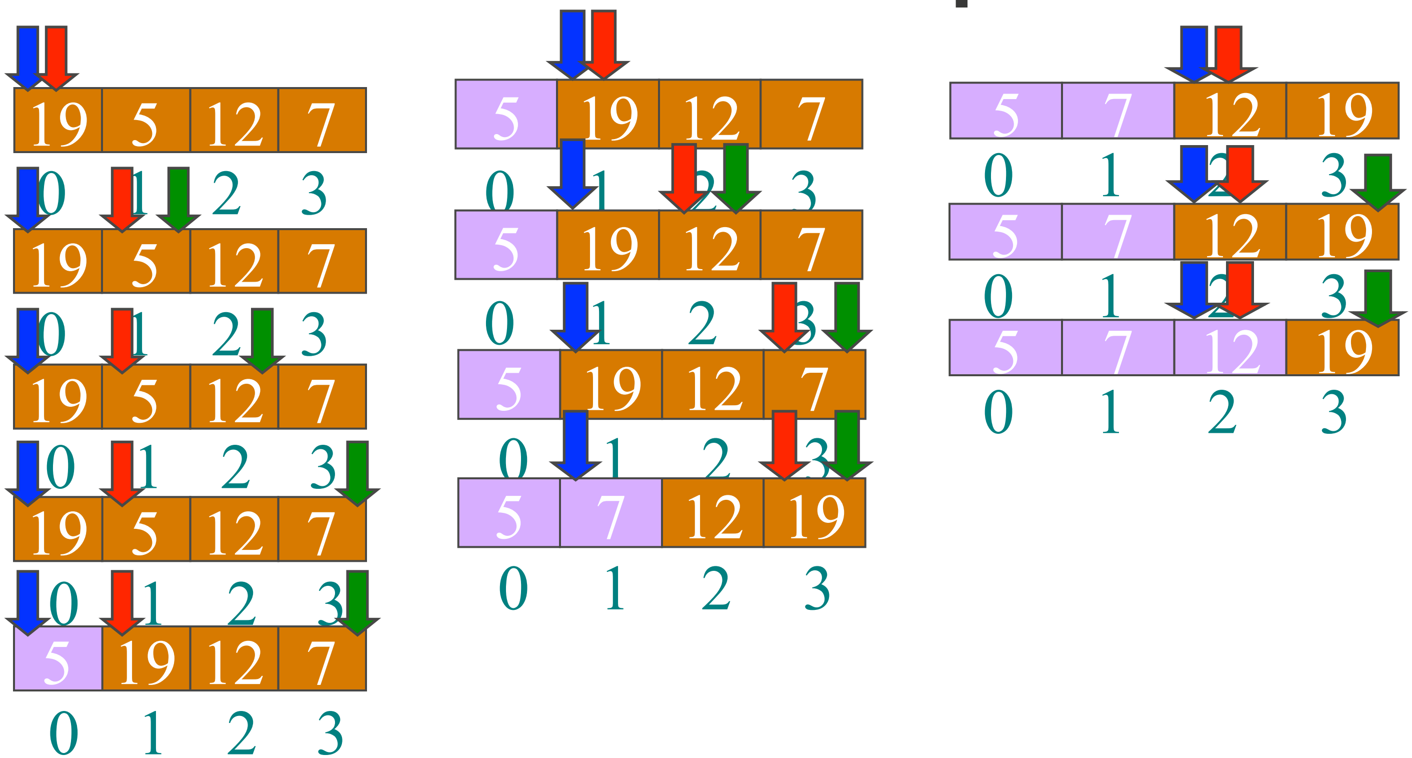




Selection Sort – Example

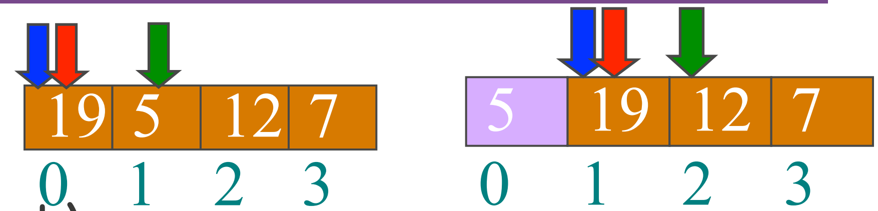


Selection Sort – Example

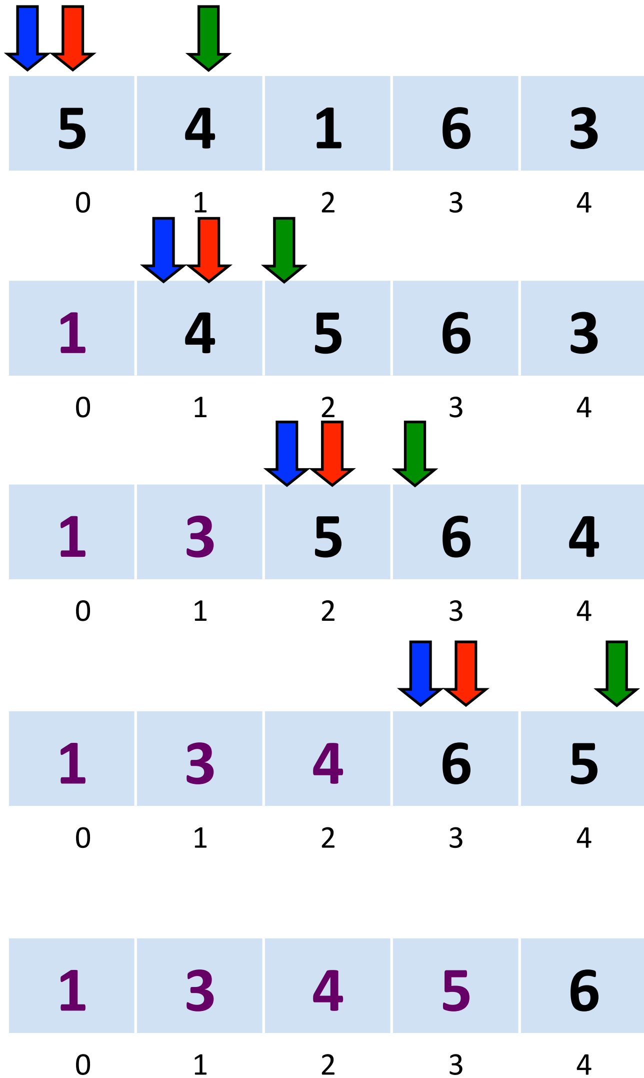


In terms of implementation: everything to the left of the (blue) mark is sorted, is in its final position and grows by one after each traversal

Selection Sort: Code

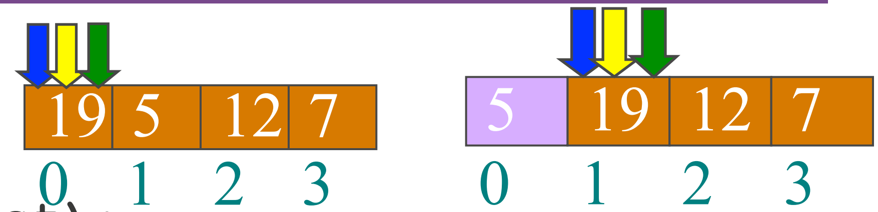



```
def selection_sort(the_list):
    n = len(the_list)
    for mark in range(n-1):
        min_index = find_min(the_list,mark)
        swap(the_list, mark, min_index)


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: Code




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

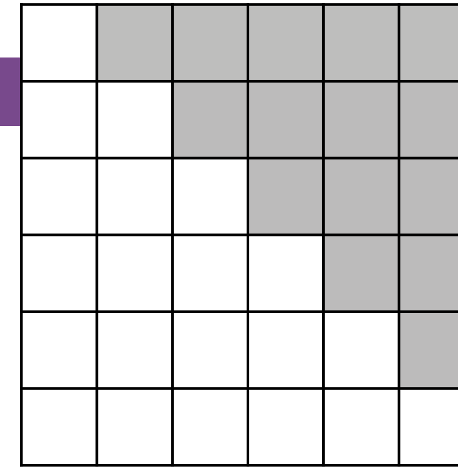
n-1 times

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

n-mark-1 times each

cons tant

Selection Sort: Time complexity



- **The inner loop**
  - Always runs for:
    $(n-1) + (n-2) + (n-3) + \ldots + 1 = n*(n-1)/2 = (n^2 - n)/2$
  - The comparison is always performed
  - The swap is always performed once per iteration
  - This only affects the constants, so $O(n^2)$
- **Any properties of the list elements that affect big O?**
  - In this case: that stop any of the two loops early?
- **No! This tells you what?**
  - best = worst
- **Same complexity as bubble sort BUT usually faster:**
  - Fewer swaps in average translate in a smaller k

Properties of sorting algorithms

Algorithm	Best case	Worst case	Stable	Incremental
Bubble Sort	$O(n^2)$	$O(n^2)$	Yes (strict)	Yes (add to front)
Bubble Sort II	$O(n)$	$O(n^2)$	Yes (strict)	Yes (add to front)
Selection Sort	$O(n^2)$	$O(n^2)$		

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

Is this Selection Sort incremental/stable?

- **Consider again the sorted list**

3	6	10	14	18	20

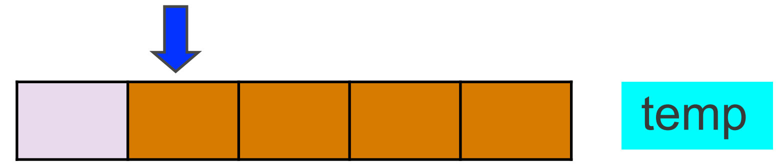
- **If we now receive element 13, can Selection Sort handle it incrementally?**
- **If we appended to the end:**
  - The list will remain unchanged for the first 3 iterations
  - Even when the mark arrives to the correct position for 13 we have not finished (number 14 is now at the end of the list!)
  - 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 in final position!)
- **Is it stable?**
  - No! we are swapping non-consecutive elements

Properties of sorting algorithms

Algorithm	Best case	Worst case	Stable	Incremental
Bubble Sort	$O(n^2)$	$O(n^2)$	Yes (strict)	Yes (add to front)
Bubble Sort II	$O(n)$	$O(n^2)$	Yes (strict)	Yes (add to front)
Selection Sort	$O(n^2)$	$O(n^2)$	No	No

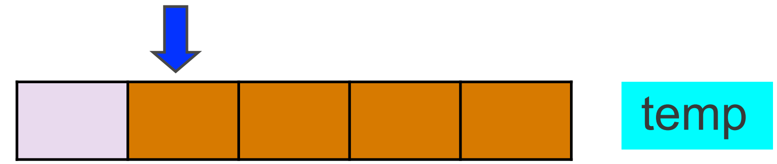
Insertion Sort

Insertion Sort



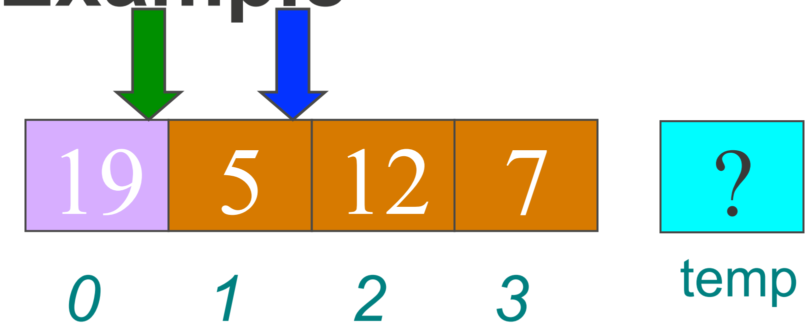
- **Main idea:**
  - Split the list into
    - Part S which is already sorted (initially one element)
    - Part U which is unsorted
  - Extend S by taking any element from U and inserting it in S maintaining the order (use addSorted)
- **For every element in U:**
  - Store the first unsorted value X 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: Invariant



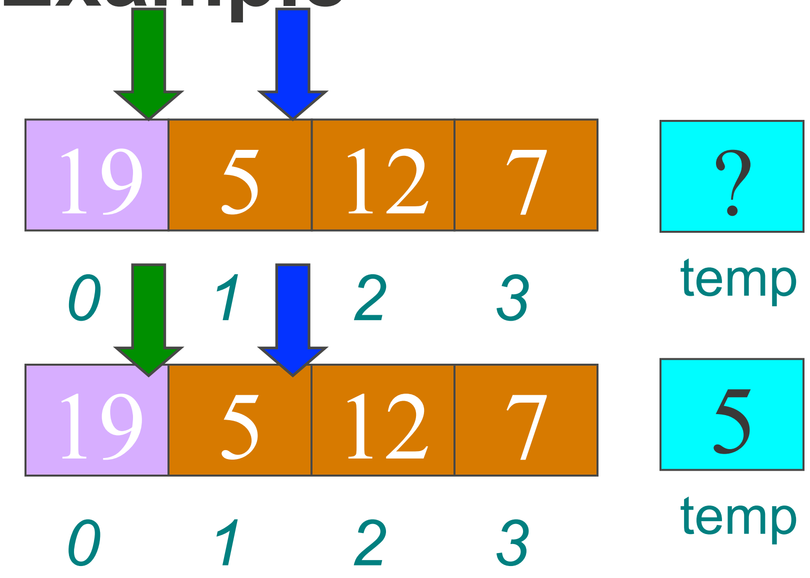
- **In terms of implementation: :**
  - I will set up on my first unsorted element
  - Everything to its left is sorted and it grows by one in each iteration
- **BUT that's only part of the list**
- **These elements might not yet be in their final position:**
  - Others may move in between them later

Insertion Sort – Example



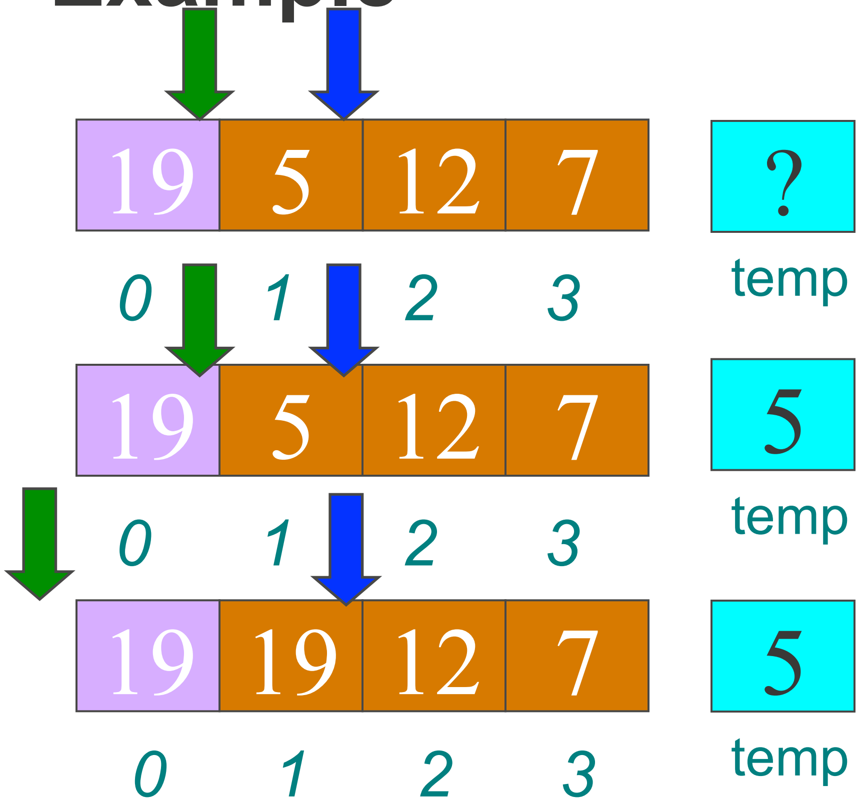
- Store unsorted in temp
- Shift bigger to right
- Store temp into freed

Insertion Sort – Example



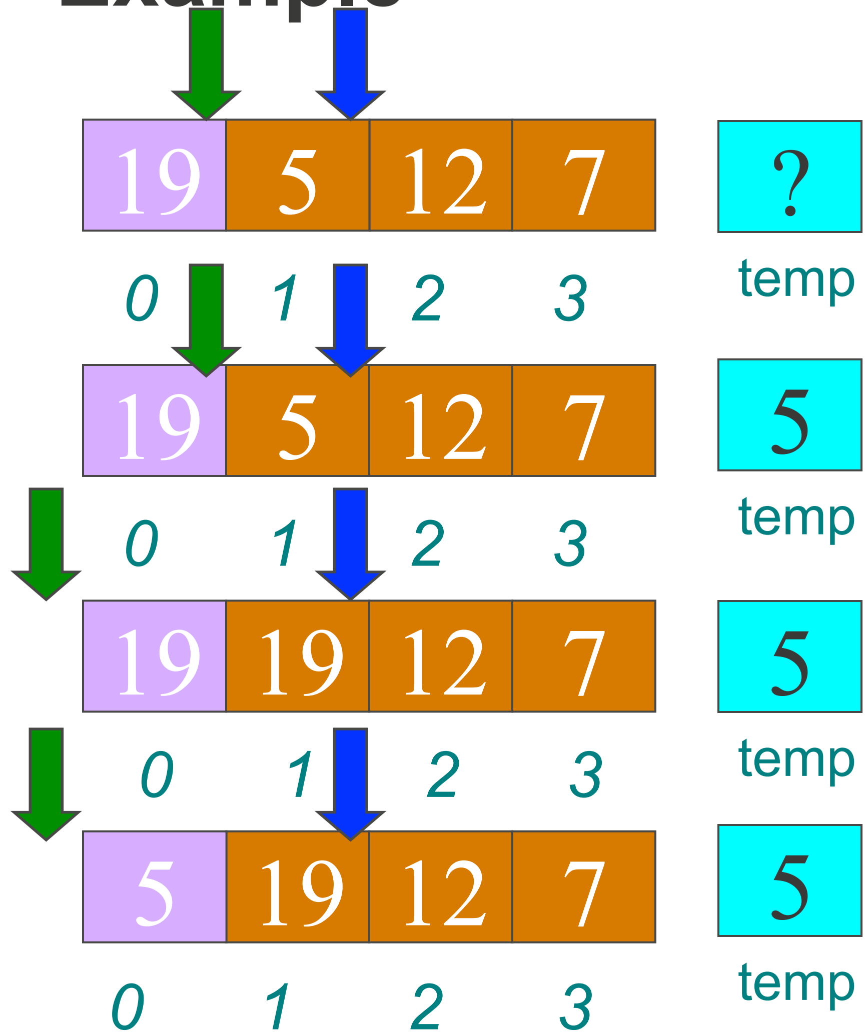
- **Store unsorted in temp**
- **Shift bigger to right**
- **Store temp into freed**

Insertion Sort – Example



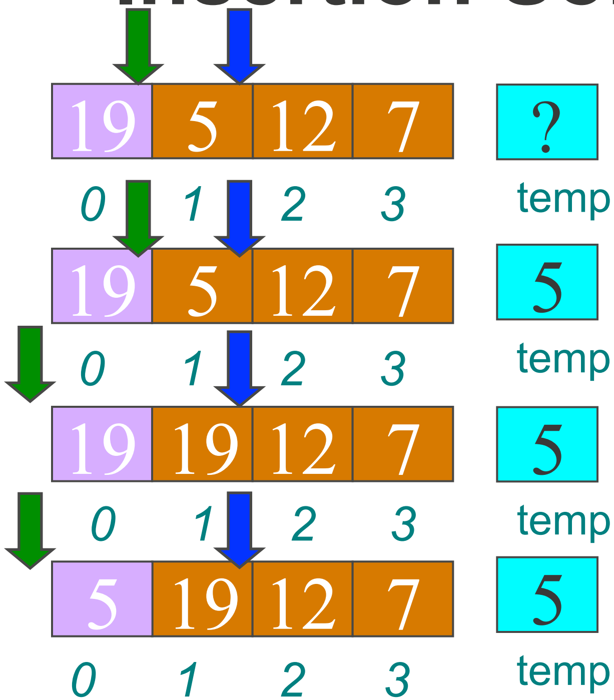
- **Store unsorted in temp**
- **Shift bigger to right**
- **Store temp into freed**

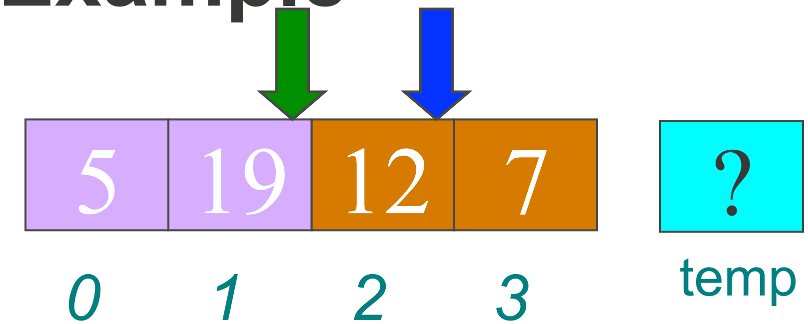
Insertion Sort – Example



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

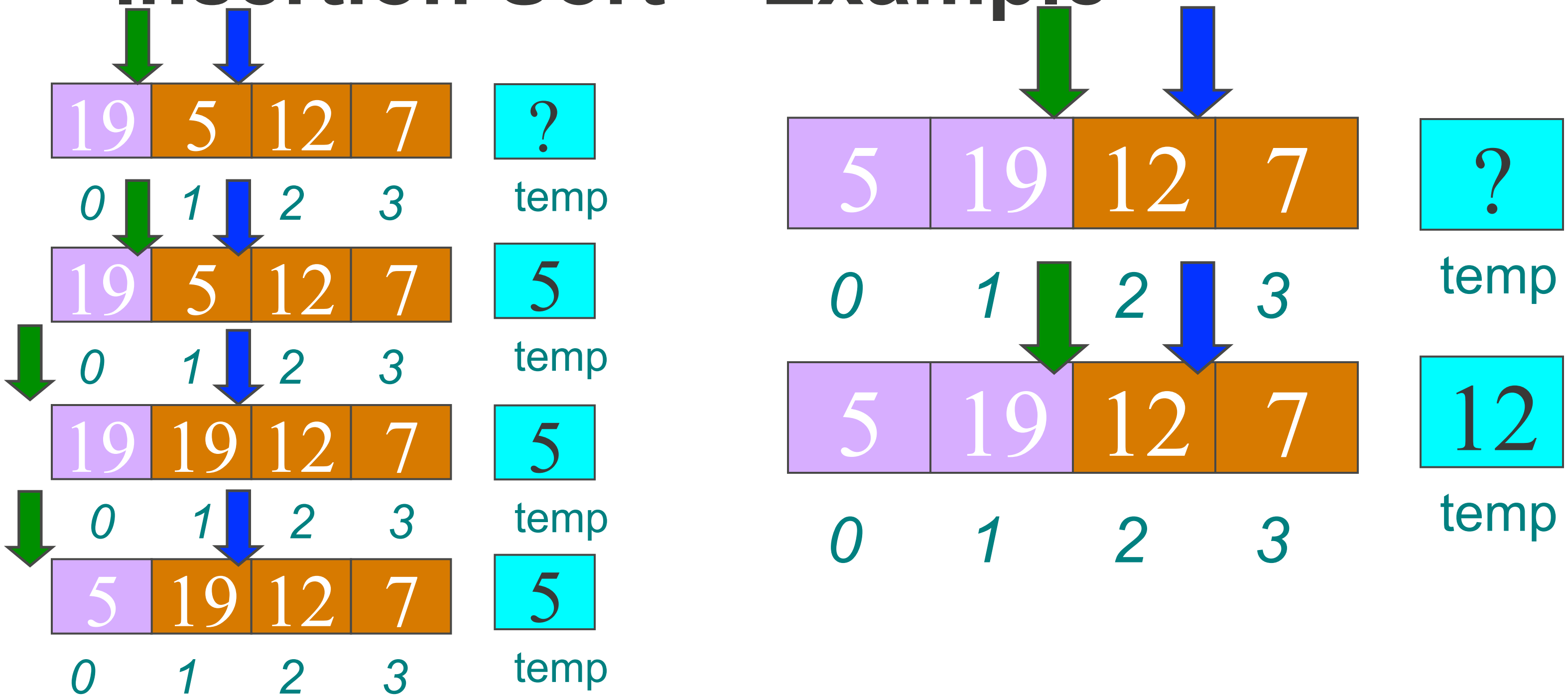
Insertion Sort – Example





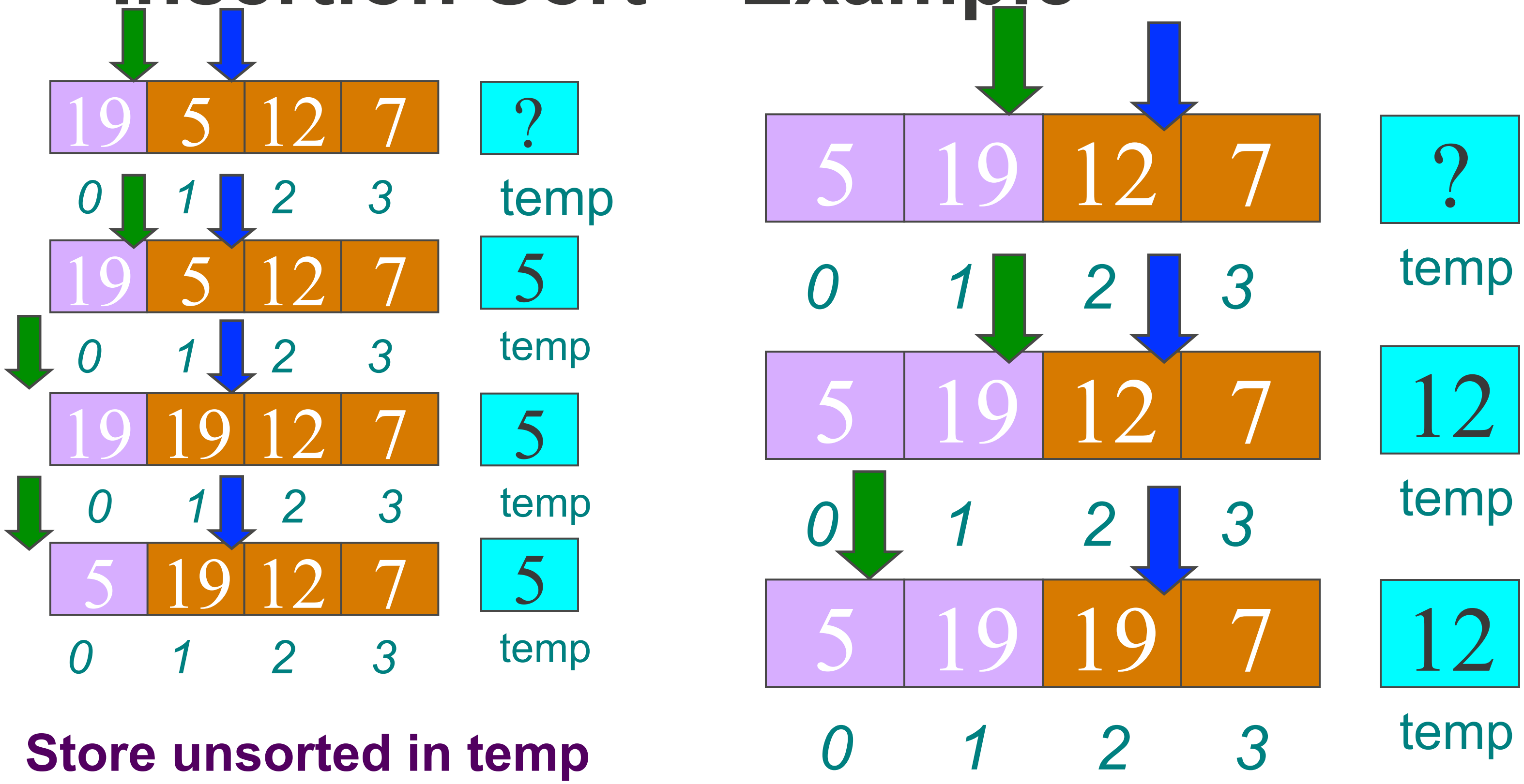
- **Store unsorted in temp**
- **Shift bigger to right**
- **Store temp into freed**

Insertion Sort – Example



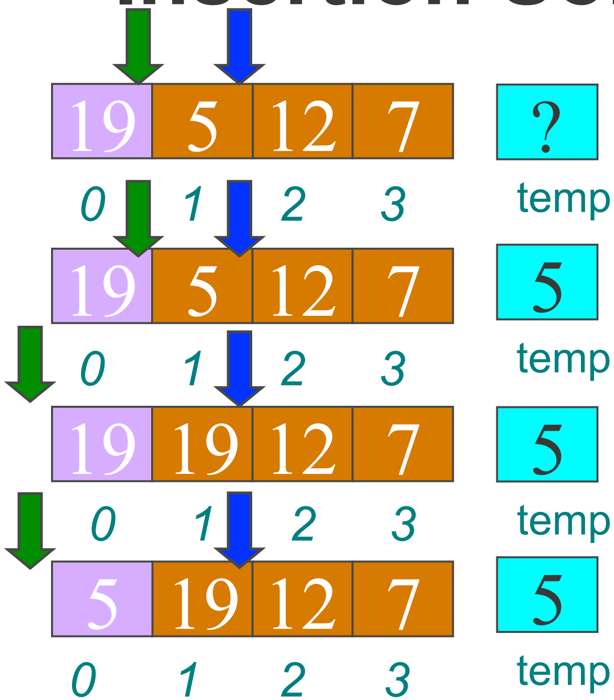
- **Store unsorted in temp**
- **Shift bigger to right**
- **Store temp into freed**

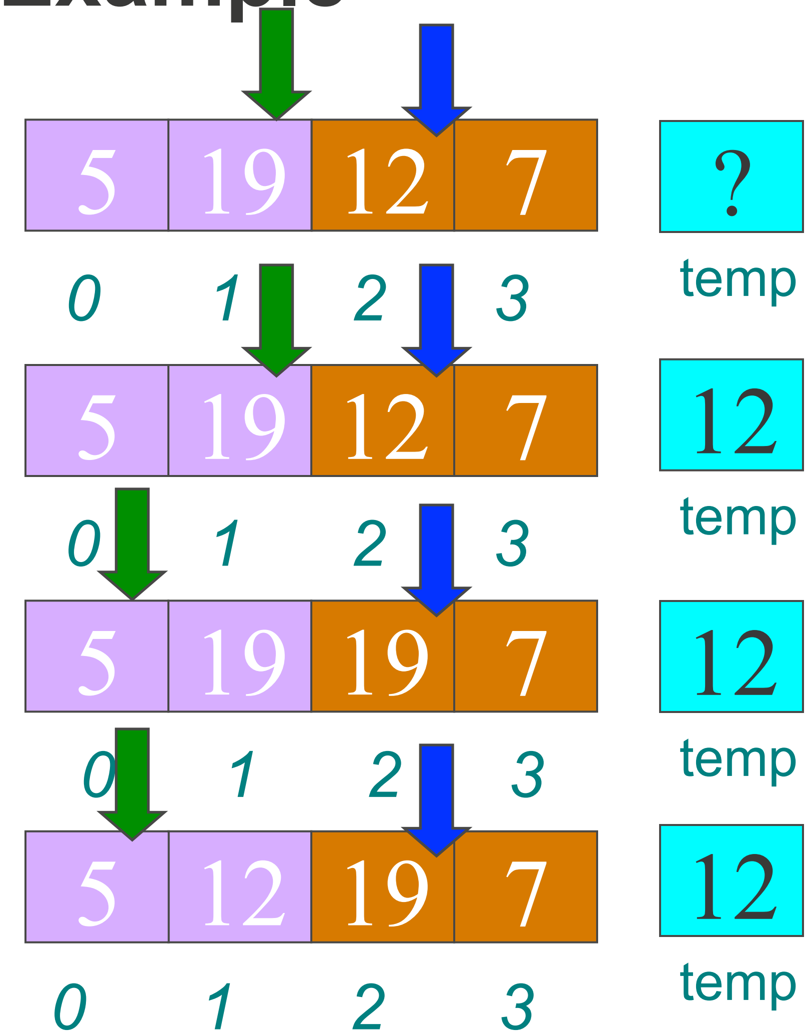
Insertion Sort – Example



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

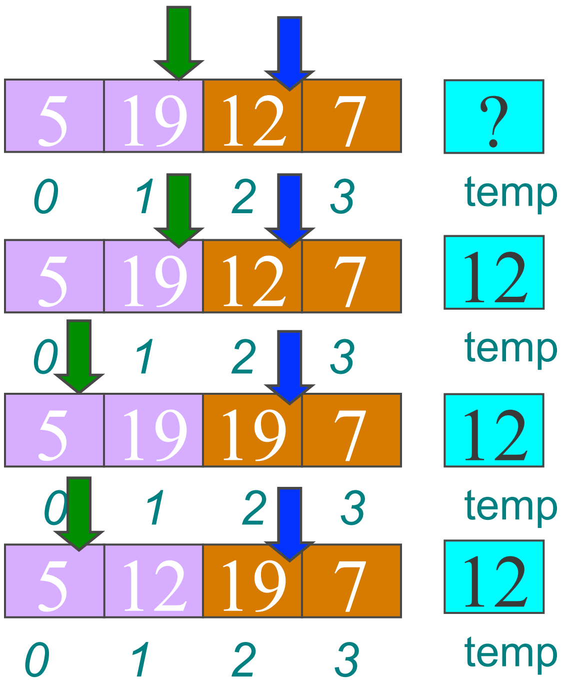
Insertion Sort – Example

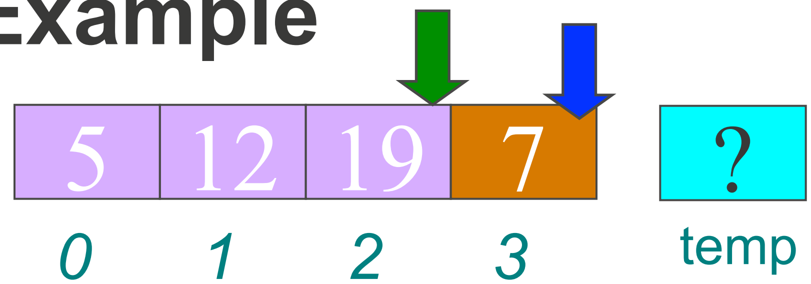




- Store unsorted in temp
- Shift bigger to right
- Store temp into freed

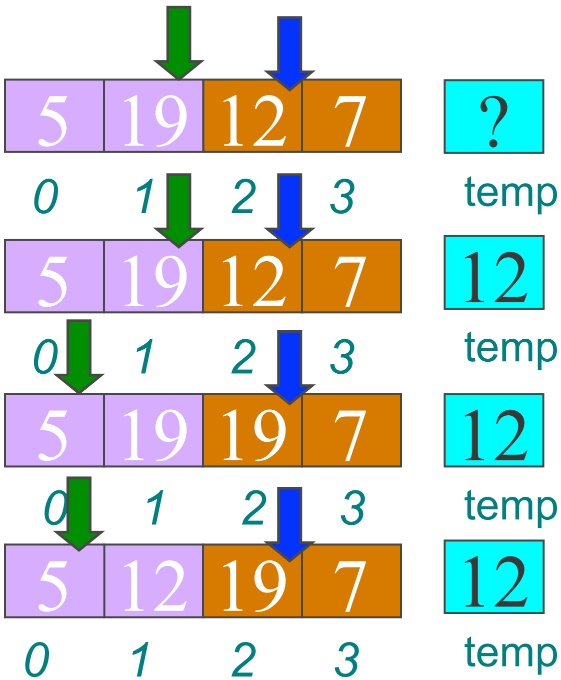
Insertion Sort – Example

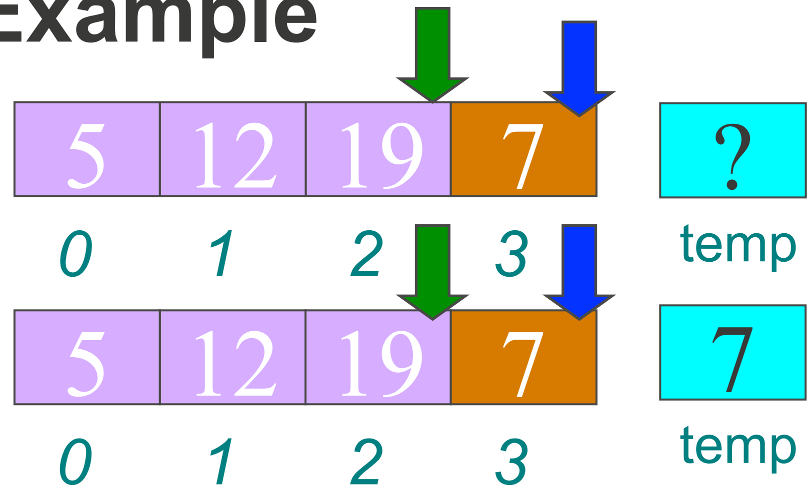




- **Store unsorted in temp**
- **Shift bigger to right**
- **Store temp into freed**

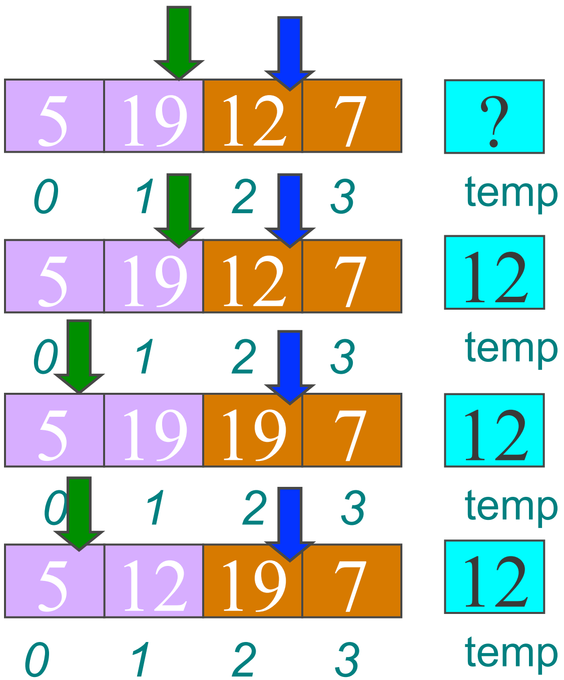
Insertion Sort – Example

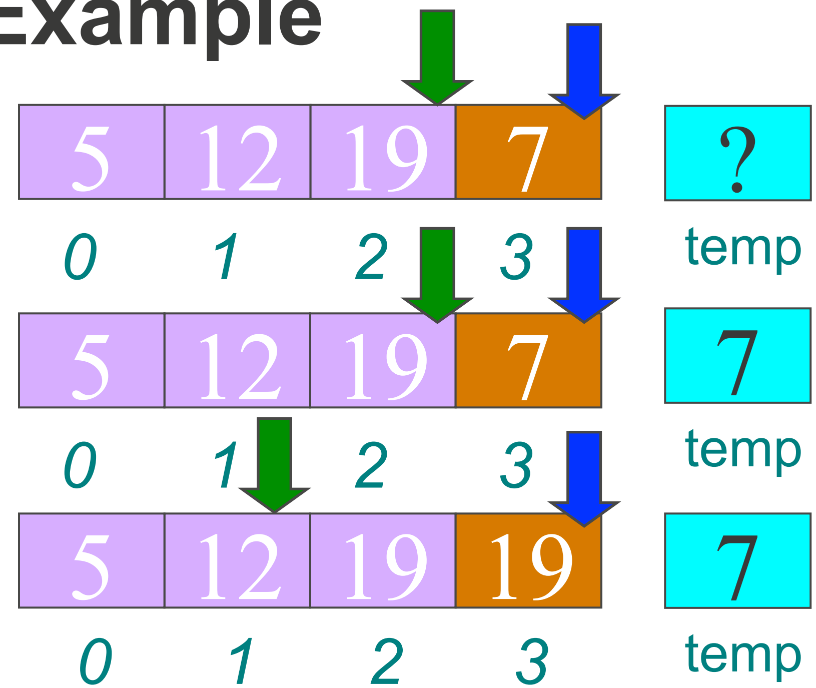




- **Store unsorted in temp**
- **Shift bigger to right**
- **Store temp into freed**

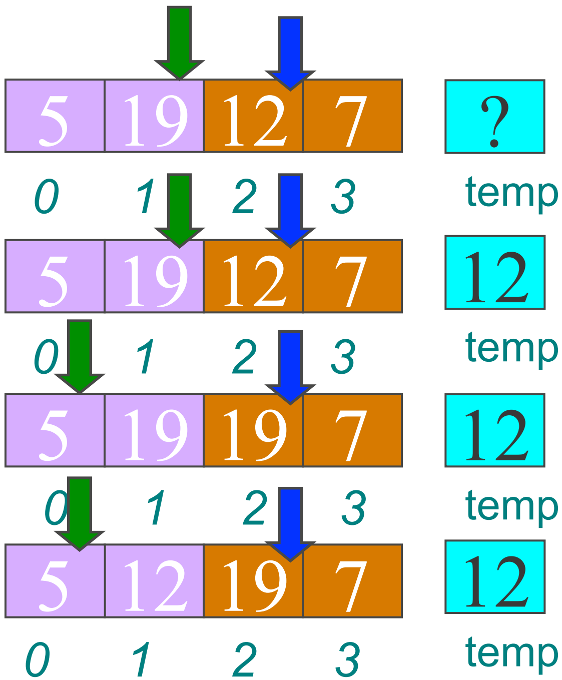
Insertion Sort – Example

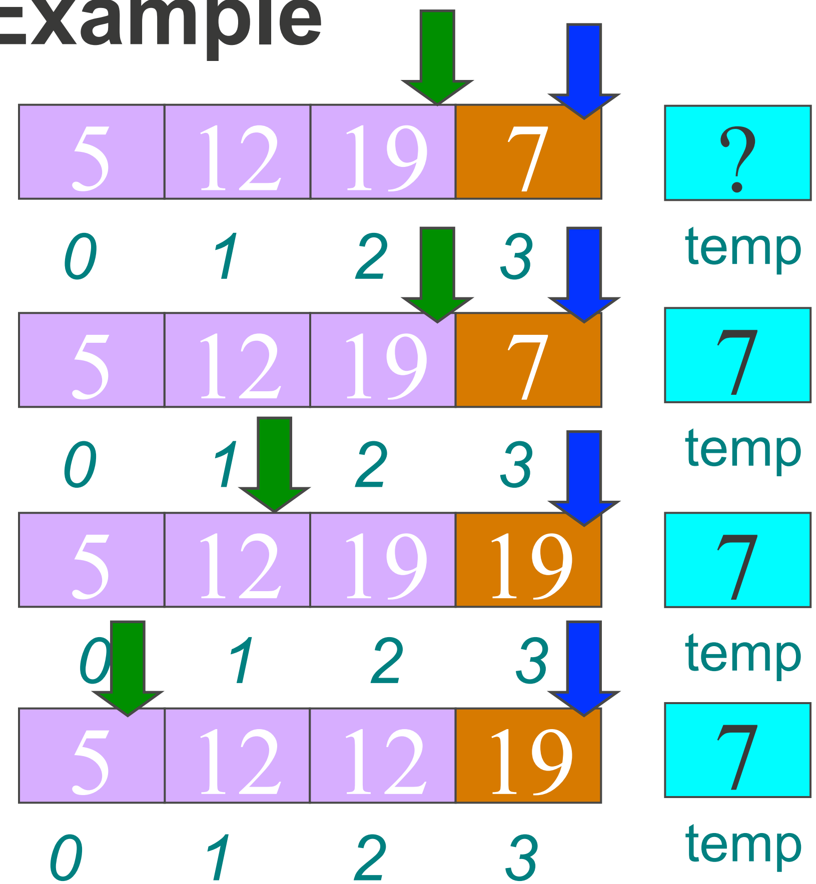




- **Store unsorted in temp**
- **Shift bigger to right**
- **Store temp into freed**

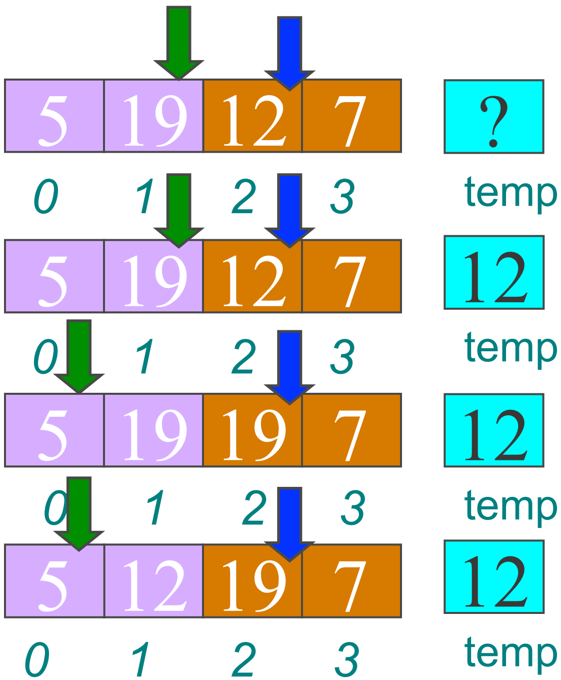
Insertion Sort – Example

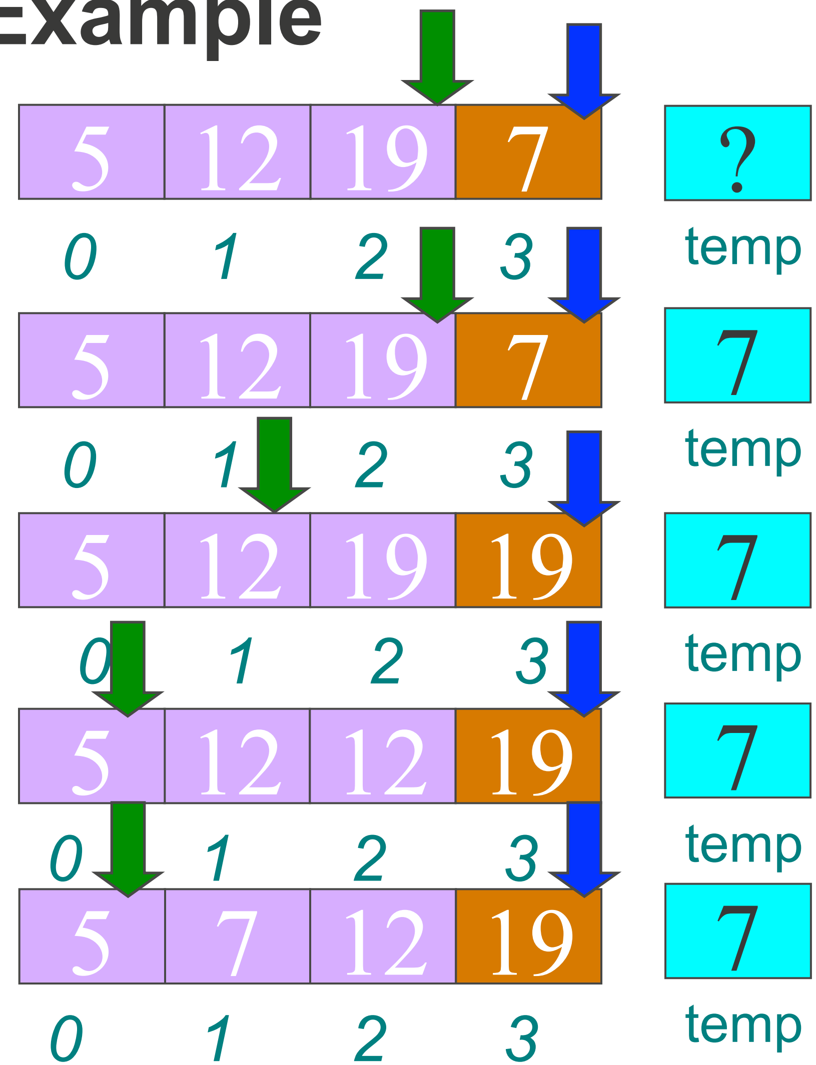




- **Store unsorted in temp**
- **Shift bigger to right**
- **Store temp into freed**

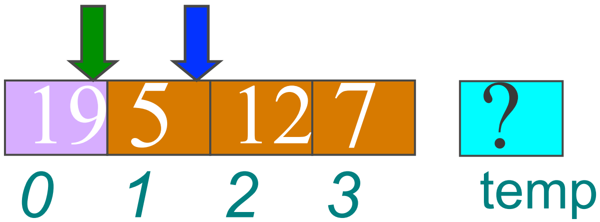
Insertion Sort – Example



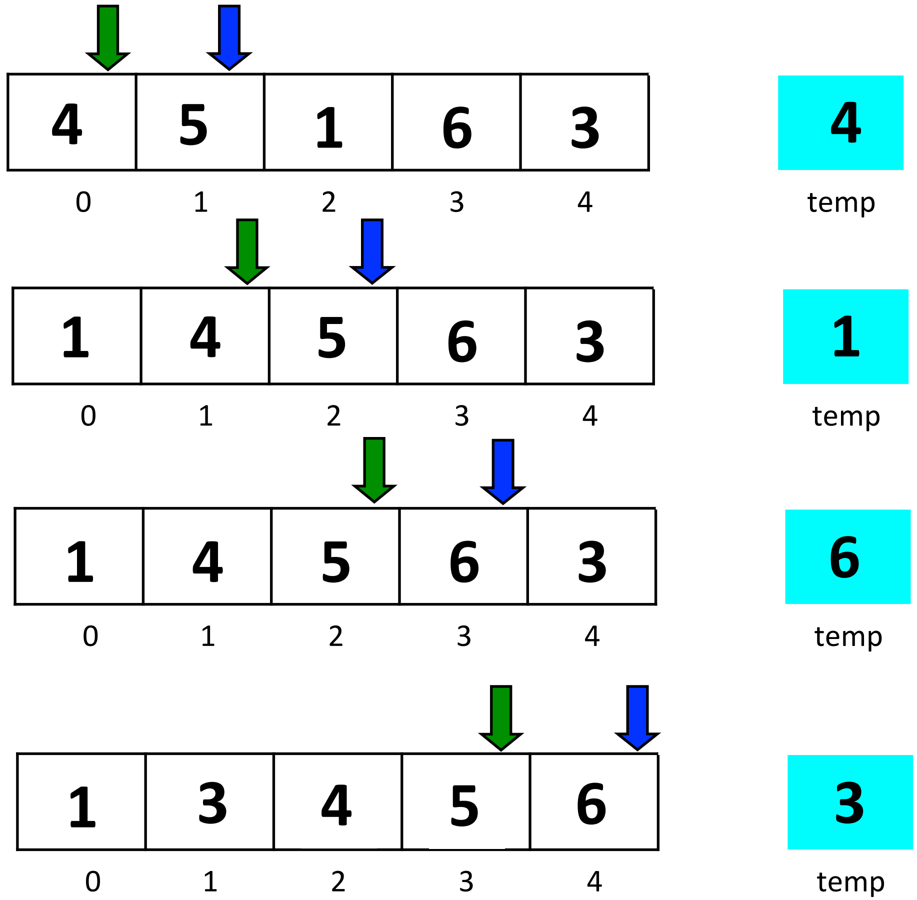


- **Store unsorted in temp**
- **Shift bigger to right**
- **Store temp into freed**

Insertion Sort: Code

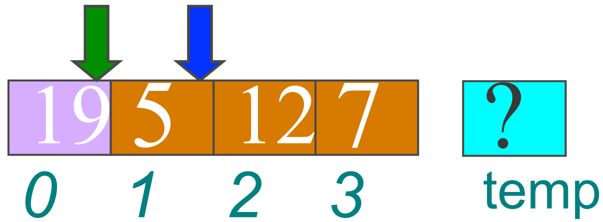



```
def insertion_sort(the_list):
    n = len(the_list)
    for mark in range(1,n):
        temp = the_list[mark]
        i = mark - 1
        while i>=0 and the_list[i] > temp:
            the_list[i+1] = the_list[i]
            i-= 1
        the_list[i+1] = temp
```




```
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
```

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
```

n-1 times

??

fixed

Insertion Sort: Time complexity

- **Can we stop any of the two loops early?**
  - Yes, the second one, when the element is already bigger
- **This already tells you what?**
  - best ≠ worst
- **Worst case?**
  - Every element needs to be shuffled to the left when inserting: the list is sorted in reverse order
  - This means $O(n^2)$ – two nested loops both dependent on n, with the inner one performing a fixed amount of steps
- **Best case?**
  - No element needs to be shuffled when inserting: the list is already sorted
  - This means O(n) – one loop dependent on n and performing a fixed amount of steps

Properties of sorting algorithms

Algorithm	Best case	Worst case	Stable	Incremental
Bubble Sort	$O(n^2)$	$O(n^2)$	Yes (strict)	Yes (add to front)
Bubble Sort II	$O(n)$	$O(n^2)$	Yes (strict)	Yes (add to front)
Selection Sort	$O(n^2)$	$O(n^2)$	No	No
Insertion Sort	$O(n)$	$O(n^2)$		

Insertion Sort: Time complexity

- **Usually faster than bubble and selection sort, especially for "almost sorted" lists**
- **Can you figure out why?**
  - What do you avoid in that case?

- **It is however slower than selection sort if our write access to memory is slow**
- **Can you figure out why?**
  - What do you do in one and not in the other?

Is this Insertion Sort incremental/stable?

- **Consider again the sorted list**

3	6	10	14	18	20

- **If we now receive element 13, can Insertion Sort handle it incrementally?**
- **If we appended to the end AND put the mark at last sorted:**
  - 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
- **Is it stable?**
  - Yes, but changing > by >= would make it not stable

Properties of sorting algorithms

Algorithm	Best case	Worst case	Stable	Incremental
Bubble Sort	$O(n^2)$	$O(n^2)$	Yes (strict)	Yes (add to front)
Bubble Sort II	$O(n)$	$O(n^2)$	Yes (strict)	Yes (add to front)
Selection Sort	$O(n^2)$	$O(n^2)$	No	No
Insertion Sort	$O(n)$	$O(n^2)$	Yes (strict)	Yes (add to back)

Points to keep in mind

- Big-O gives an upper bound. May be much larger than the actual one
- The input that gives the worst case may be very unlikely
- Big-O ignores constants. In practice they may be very large
- If a program is used only a few times, then the actual running time may not be a big factor in the overall costs
- If a program is only used on small inputs, the growth rate of the running time may be less important than other factors
- A complex but efficient algorithm can be less desirable than a simpler one
- Other criteria: In numerical algorithms, other properties (like stability and incrementally) can be as important as efficiency
- The average case is always between the best and the worst cases

Summary

- **After these two lectures you are now able to:**
  - Compute the Big O of simple functions (best and worst case)
  - Implement, use and modify the following sorting algorithms:
    - Bubble Sort (seen in the prac)
    - Selection Sort
    - Insertion Sort
  - Determine important invariants of the sorting algorithms and use them to improve the algorithms
  - In particular, reason about the stability and incrementality of the sorting algorithms