Intermezzo – correcting code

(and preventing common mistakes)

Code example 1

```
def fahrenheitToCelsius(fahrenheit):
   celsius = (5/9)*(fahrenheit - 32)
   print(celsius)

def isFreezing(fahrenheit):
   if(fahrenheitToCelsius(fahrenheit) <= 0):
      return true
   else:
      return false

print(isFreezing(100)) #100°F
print(isFreezing(0))   #0°F
```

Code example 1

```
def fahrenheitToCelsius(fahrenheit):
   celsius = (5/9)*(fahrenheit - 32)
   print(celsius)

def isFreezing(fahrenheit):
   if(fahrenheitToCelsius(fahrenheit) <= 0):
      return true
   else:
      return false

print(isFreezing(100)) #100°F
print(isFreezing(0))   #0°F
```

Code example 1 - corrected

```
def fahrenheitToCelsius(fahrenheit):
   celsius = (5/9)*(fahrenheit - 32)
   return celsius

def isFreezing(fahrenheit):
   if(fahrenheitToCelsius(fahrenheit) <= 0):
      return True
   else:
      return False

print(isFreezing(100)) #100°F
print(isFreezing(0))   #0°F
```

Code example 2

```
def find(list, element):
    for i in range(list):
         if (list[i] == element):
             return i
         else:
             return 'Element not found in list'

list = [1,4,7,6,2,12,8,9]
print(find(list, 7))
```

Code example 2

```
def find(list, element):
    for i in range(list):
         if (list[i] == element):
             return i
         else:
             return 'Element not found in list'

list = [1,4,7,6,2,12,8,9]
print(find(list, 7))
```

Code example 2 - corrected

```
def findElement(mylist, element):
    for i in range(len(mylist)):
         if (list[i] == element):
             return i
    return 'Element not found in list'

L = [1,4,7,6,2,12,8,9]
print(findElement(L, 7))
```

Code example 3

```
def reverseList(list):
     reversed = ()
     for i in range(len(list), 0, -1):
         reversed.append(list[i])
         return reversed

list = (0,1,2,3,4,5)
print(reverseList(list))
```

Code example 3

```
def reverseList(list):
     reversed = ()
     for i in range(len(list), 0, -1):
         reversed.append(list[i])
         return reversed

list = (0,1,2,3,4,5)
print(reverseList(list))
```

Code example 3 - corrected

```
def reverseList(mylist):
     reversed = list()
     for i in range(len(mylist)-1, -1, -1):
         reversed.append(mylist[i])
     return reversed

L = [0,1,2,3,4,5]
print(reverseList(L))
```

Beyond lists

Tuples, sets and dictionaries

Tuples

A tuple is an **ordered** sequence of **immutable** objects. Tuples use parentheses.

Example:

```
tup1 = ('physics', 'chemistry', 1997, 2000)
tup2 = (1, 2, 3, 4, 5 )
tup3 = "a", "b", "c", "d";
print(tup1)
print(tup2)
print(tup3)
```

Output:

```
('physics', 'chemistry', 1997, 2000)
(1, 2, 3, 4, 5)
('a', 'b', 'c', 'd')
```

Tuples
a detail…

Creating a tuple with only one element

```
x = (9)
print(type(x))
x = (9,)
print(type(x))
```

Output:

```
<class 'int'>
<class 'tuple'>
```

Tuples
basic operations

Expression	Results	Description
`len((1, 2, 3))`	`3`	Length
`(1, 2, 3) + (4, 5, 6)`	`(1, 2, 3, 4, 5, 6)`	Concatenation
`('Hi!',) * 3`	`('Hi!', 'Hi!', 'Hi!')`	Repetition
`3 in (1, 2, 3)`	`True`	Membership
`for x in (1, 2, 3):` `  print x,`	`1 2 3`	Iteration

Tuples
basic operations

Example:

```
tup1 = (12, 34.56);
tup1[0] = 100;
```

Output:

```
TypeError: 'tuple' object does not support item assignment
```

Tuples are immutable!

Sets

A set is an **unordered** collection of **distinct** items.

- unordered: there is no particular order
- distinct: any item in the set appears once

Example:

```
x = {'a','s','d','s'}
print(x)
```

Output:

```
{'a', 'd', 's'}
```

Sets

Sets are mutable (i.e. can be changed).
They are used when a collection of unique objects is wanted.

Example:

```
x = set([3, 1, 2, 1])
print(x)
```

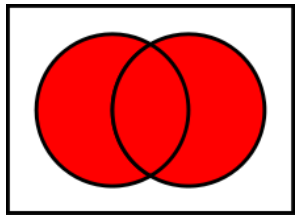
Output:

```
{1, 2, 3}
```

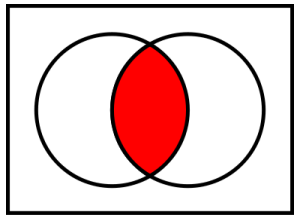
Sets

basic operations

In mathematics, set operations include union, intersection, add and remove. All of these (and others) exist in Python.



A ∪ B



A ∩ B

Sets

basic operations

In mathematics, set operations include union, intersection, add and remove. All of these (and others) exist in Python.

Example:

```
ten = set(range(10))
lows = {0, 1, 2, 3, 4}
odds = {1, 3, 5, 7, 9}
lows.add(9)
print(lows)

print(lows.difference(odds))
print(lows.intersection(odds))
print(lows.issubset(ten))
print(lows.issuperset(odds))

lows.remove(0)
print(lows)
lows.union(odds)
print(lows)
```

Sets

basic operations

In mathematics, set operations include union, intersection, add and remove. All of these (and others) exist in Python.

Example:

```
ten = set(range(10))
lows = {0, 1, 2, 3, 4}
odds = {1, 3, 5, 7, 9}
lows.add(9)
print(lows)

print(lows.difference(odds))
print(lows.intersection(odds))
print(lows.issubset(ten))
print(lows.issuperset(odds))

lows.remove(0)
print(lows)
lows.union(odds)
print(lows)
```

Output:

```
{0, 1, 2, 3, 4, 9}
{0, 2, 4}
{9, 3, 1}
True
False
{1, 2, 3, 4, 9}
{1, 2, 3, 4, 5, 7, 9}
```

Dictionaries

A **dictionary** is a collection of values. But unlike indexes for lists, indexes for dictionaries can be of different data types, not just integers. Indexes for dictionaries are called **keys**, and a key with its value is called a **key-value** pair.

Dictionaries

A dictionary is used to **map** or **associate** things you want to store to keys.

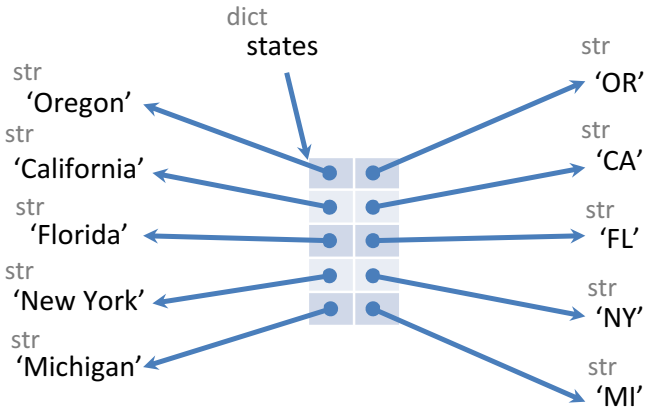
Example:

```
# create a mapping of state to abbreviation
states = {
    'Oregon': 'OR',
    'Florida': 'FL',
    'California': 'CA',
    'New York': 'NY',
    'Michigan': 'MI'
}
```

Dictionaries

conceptually

A dictionary is used to **map** or **associate** things you want to store to keys.



Dictionaries

Example:

```
# create a mapping of state to abbreviation
states = {'Oregon':'OR', 'Florida':'FL',
   'California':'CA', 'New York':'NY', 'Michigan':'MI'}

print("Michigan's abbreviation is: ", states['Michigan'])
print("Florida's abbreviation is: ", states['Florida'])
```

Output:

```
Michigan's abbreviation is:  MI
Florida's abbreviation is:  FL
```

Dictionaries
an example

```
birthdays = {'Alice': 'Apr 1', 'Bob': 'Dec 12', 'Carol': 'Mar 4'}
while True:
   print('Enter a name: (blank to quit)')
   name = input()
   if name == '':
      break
   if name in birthdays:
      print(birthdays[name] + ' is the birthday of ' + name)
   else:
      print('I do not have birthday information for ' + name)
      print('What is their birthday?')
      bday = input()
      birthdays[name] = bday
      print('Birthday database updated.')
```

Dictionaries
some built-in methods

Method	Description
`dict.clear()`	Removes all elements of dictionary *dict.*
`dict.get(key)`	For *key* key, returns corresponding value.
`dict.get(key, v)`	For *key* key, returns value or default *v* if the key is not in *dict.*
`dict.keys()`	Returns a list of dictionary *dict*'s keys.
`dict.items()`	Returns a list of *dict*'s (key, value) tuple pairs.
`dict.pop(key)`	Removes *key* from dictionary *dict* and returns value associated to *key.*
`dict.update(d2)`	Updates dictionary *dict* with content of dictionary *d2.*

Reading and Writing Files

Reading and writing files
opening a file

```
file = open(file_name [, access_mode])
```

access_mode determines the mode in which the file has to be opened, i.e., read, write, append, etc.

Modes	Description
r	Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
w	Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
a	Opens a file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.

Reading and writing files

reading a file

The with statement:

```
with open(<<filename>>, <<mode>>) as <<variable>>:
   <<block>>
```

An example:

```
with open('file_example.txt', 'r') as file:
   contents = file.read()
print(contents)
```

Use this construction when you want to store the content of the file into a single string.

Reading and writing files

reading a file

Another example:

```
with open('file_example.txt', 'r') as file:
   first_ten_chars = file.read(10)
   remainder = file.read()

print('First ten characters: ', first_ten_chars )
print('The rest of the file: ', remainder)
```

`file.read(10)` moves the file cursor to character 10, the next read call will start at character 11.

Reading and writing files

reading a file with readlines()

Example text file:

```
First Line of text
Second Line of text
Third Line of text
```

Code:

```
with open('file_example.txt', 'r') as file:
   lines = file.readlines()
print(lines)
```

Output:

```
['First Line of text\n', 'Second Line of text\n', 'Third Line of
text']
```

`readlines()` reads the content of a file and stores it into a list of strings.

Reading and writing files

reading a file

Yet another way…

```
f = open('file_example.txt','r')
allLines = list()
for line in f:
   allLines.append(line)
print(allLines)
```

Output:

```
['First Line of text\n', 'Second Line of text\n',
'Third Line of text']
```

Reading and writing files

writing to a file

Create (or overwrite) a new file

```
with open('topics.txt', 'w') as output_file:
   output_file.write('scripting languages')
```

Add to a file:

```
with open('topics.txt', 'a') as output_file:
   output_file.write('embedded systems')
```

Algorithms

Recursion

Recursion

Definition

- Recursion is a method for solving problems that involves breaking a problem down into smaller and smaller sub-problems until you get to a small enough problem that can be solved trivially.
- Usually recursion involves a function calling itself.
- Recursion allows us to write elegant solutions to problems that may otherwise be very difficult to program.

*source: http://interactivepython.org/

Recursion – A simple example

the sum of a list of numbers

Classic solution:

```
def listsum(aList):
   aSum = 0
   for el in aList:
      aSum = aSum + el
   return aSum

print(listsum([2,3,6,7,10]))
```

How can we use recursion?

Recursion – A simple example

the sum of a list of numbers

Decompose the problem in smaller problems:

```
sum([2,3,6,7,10]) = 2 + sum([3,6,7,10])
sum([3,6,7,10])   = 3 + sum([6,7,10])
⋮
sum([7,10])       = 7 + sum([10])
```

General pattern:

```
sum_of_list = first_element + sum_of_restoflist
```

In proper Python syntax:

```
sum(numList)= numList[0] + sum(numList[1:])
```

Recursion – A simple example

the sum of a list of numbers

Decompose the problem in smaller problems:

```
sum([2,3,6,7,10]) = 2 + sum([3,6,7,10])
sum([3,6,7,10])   = 3 + sum([6,7,10])
⋮
sum([7,10])       = 7 + sum([10])
```

Be careful: Recursive calls should continue until the sub-list has only one element left, then the value of that element should be returned!

Recursion – A simple example

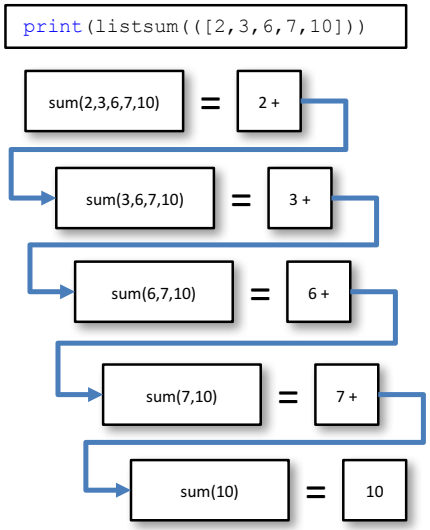
the sum of a list of numbers

```
def listsum(numList):
    if len(numList) == 1:
        return numList[0]
    else:
        return numList[0] + listsum(numList[1:])


print(listsum(([2,3,6,7,10]))
```

Recursion – A simple example

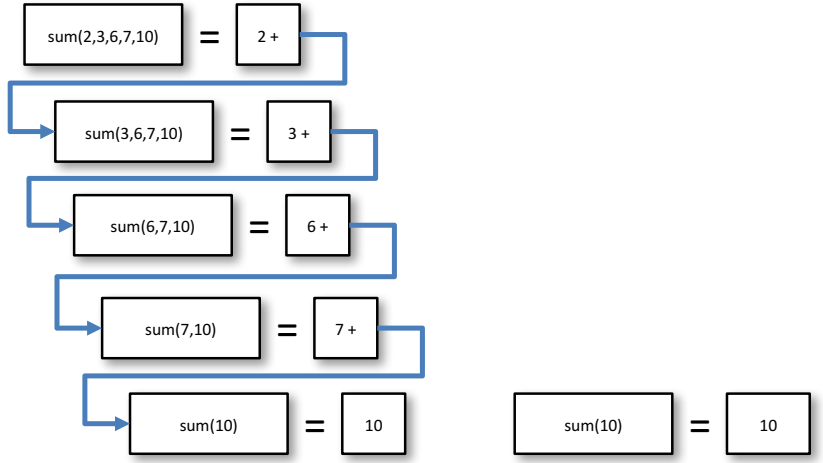
the sum of a list of numbers



Recursion – A simple example

the sum of a list of numbers

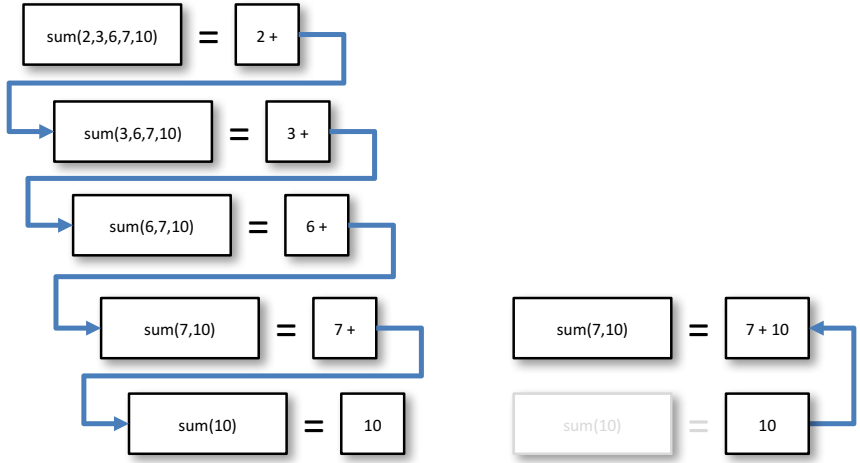
```
print(listsum([2,3,6,7,10]))
```



Recursion – A simple example

the sum of a list of numbers

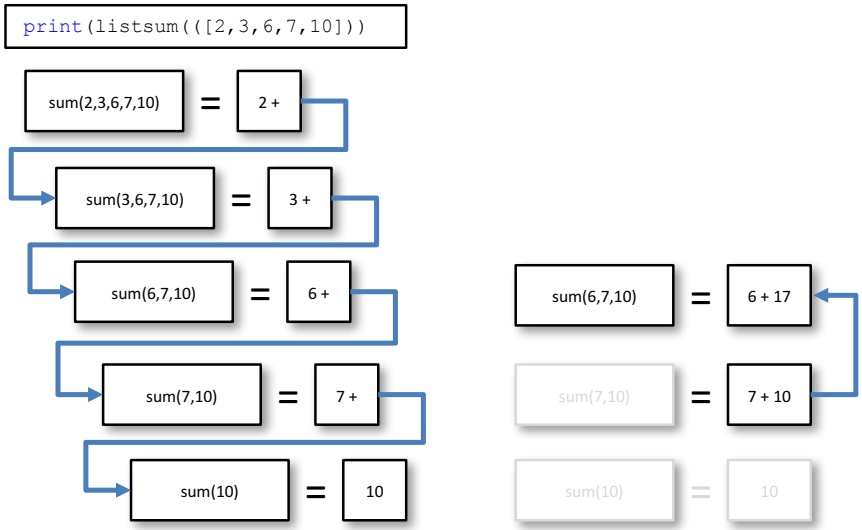
```
print(listsum([2,3,6,7,10]))
```



Recursion – A simple example

the sum of a list of numbers

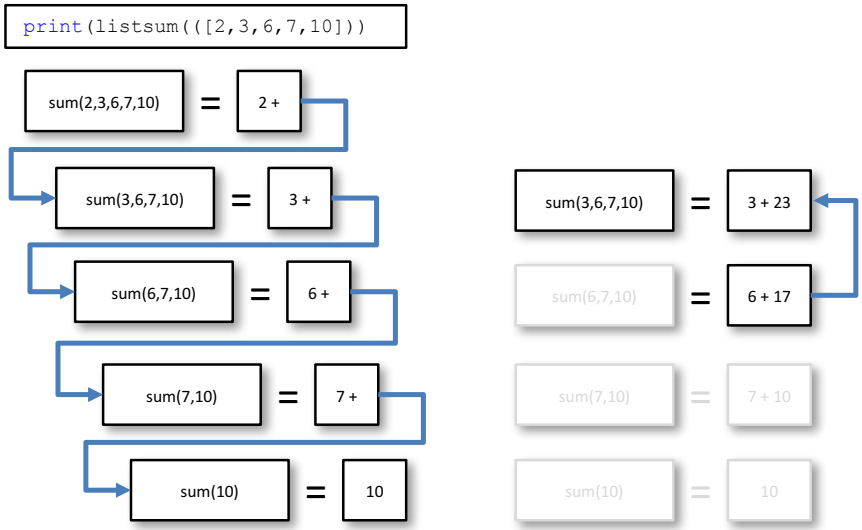
```
print(listsum([2,3,6,7,10]))
```



Recursion – A simple example

the sum of a list of numbers

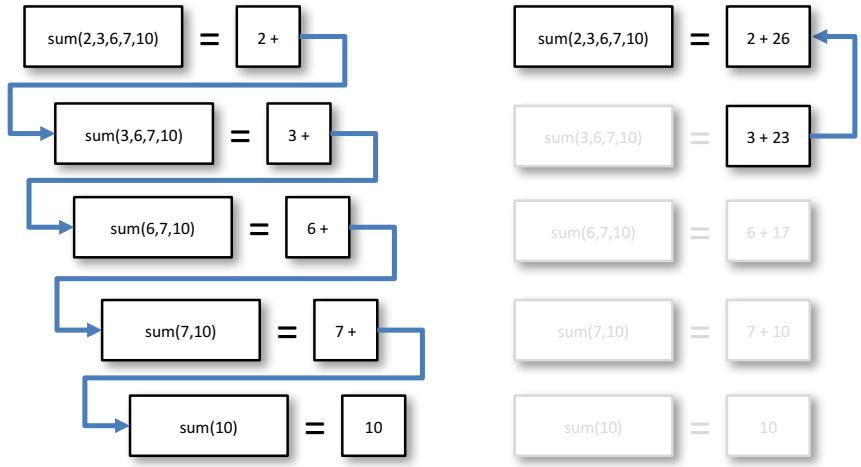
```
print(listsum([2,3,6,7,10]))
```



Recursion – A simple example

the sum of a list of numbers

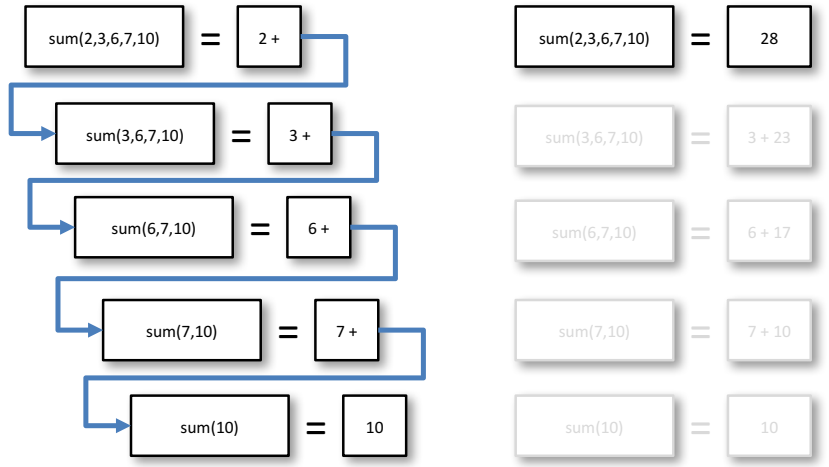
```
print(listsum([2,3,6,7,10]))
```



Recursion – A simple example

the sum of a list of numbers

```
print(listsum([2,3,6,7,10]))
```



Sorting Algorithms

Sorting algorithms

In-place vs. Not-in-place

An **in-place** algorithm transforms its input without using an auxiliary data structure. The algorithm updates an input sequence only through replacement or swapping of elements.

e.g. an in-place implementation of reversing a list:

```
def reverseinplace(mylist):
    for i in range(len(mylist)//2):
        temp = mylist[len(mylist)-i-1]
        mylist[len(mylist)-i-1] = mylist[i]
        mylist[i] = temp
    return mylist
```

Sorting algorithms

In-place vs. Not-in-place

An **In-place** algorithm transforms its input without using an auxiliary data structure. The algorithm updates an input sequence only through replacement or swapping of elements.

e.g. a not-in-place implementation of reversing a list:

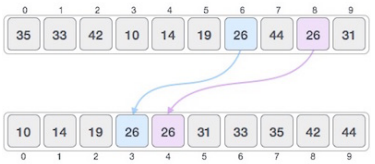
```
def reverselist(mylist):
    reversed = list()
    for i in range(len(mylist)-1, -1, -1):
        reversed.append(mylist[i])
    return reversed
```

Sorting algorithms

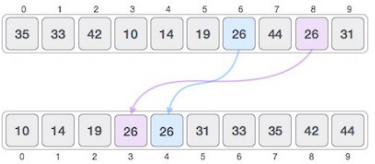
Stable vs. not-stable

A **stable** sorting algorithm maintains the relative order of records with equal values. Important if there is additional information attached to the values being sorted. Stable sorting algorithms ensure that sorting an already sorted list leaves the order of the list unchanged.

Stable



Not-stable



Sorting Algorithms

Bubble Sort

Bubble Sort

algorithm

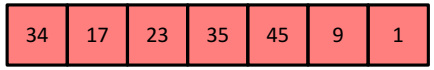
Step 1: Compare the first pair of elements in the unsorted list and swap if the first element is larger than the second.

Step 2: Compare the subsequent pairs of elements and swap if necessary, ensuring that the largest element is at the end of the list.

Step 3: Keep repeating **Step 1** and **Step 2** for one fewer elements each time until there are no more pairs to compare.

Bubble Sort
algorithm

Step 1: Compare the first pair of adjacent elements in the unsorted list and swap if the first element is larger than the first.

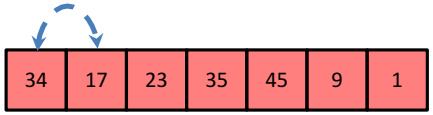


unsorted

sorted

Bubble Sort
algorithm

Step 1: Compare the first pair of adjacent elements in the unsorted list and swap if the first element is larger than the first.

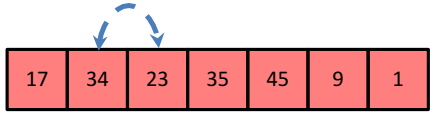


unsorted

sorted

Bubble Sort
algorithm

Step 2: Compare the subsequent pairs of elements and swap if necessary, ensuring that the largest element is at the end of the list.

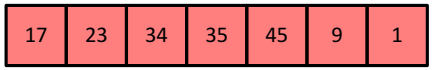


unsorted

sorted

Bubble Sort
algorithm

Step 2: Compare the subsequent pairs of elements and swap if necessary, ensuring that the largest element is at the end of the list.

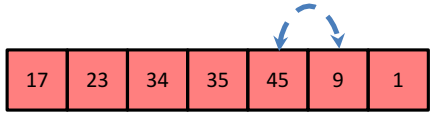


unsorted

sorted

Bubble Sort
algorithm

Step 2: Compare the subsequent pairs of elements and swap if necessary, ensuring that the largest element is at the end of the list.

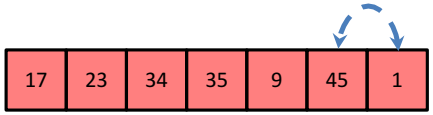


unsorted

sorted

Bubble Sort
algorithm

Step 2: Compare the subsequent pairs of elements and swap if necessary, ensuring that the largest element is at the end of the list.

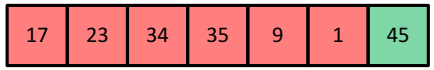


unsorted

sorted

Bubble Sort
algorithm

Step 2: Compare the subsequent pairs of elements and swap if necessary, ensuring that the largest element is at the end of the list.

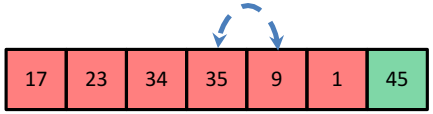


unsorted

sorted

Bubble Sort
algorithm

Step 3: Keep repeating for one fewer elements each time until there are no more pairs to compare.

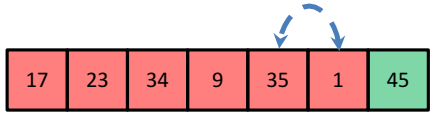


unsorted

sorted

Bubble Sort
algorithm

Step 3: Keep repeating for one fewer elements each time until there are no more pairs to compare.

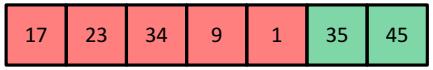


unsorted

sorted

Bubble Sort
algorithm

Step 3: Keep repeating for one fewer elements each time until there are no more pairs to compare.

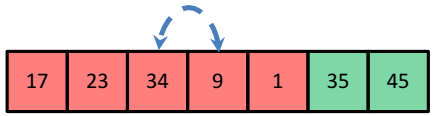


unsorted

sorted

Bubble Sort
algorithm

Step 3: Keep repeating for one fewer elements each time until there are no more pairs to compare.

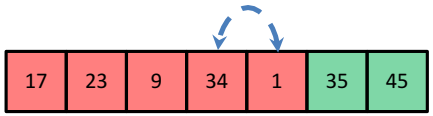


unsorted

sorted

Bubble Sort
algorithm

Step 3: Keep repeating for one fewer elements each time until there are no more pairs to compare.

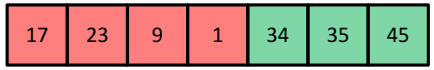


unsorted

sorted

Bubble Sort
algorithm

Step 3: Keep repeating for one fewer elements each time until there are no more pairs to compare.



unsorted

sorted

Bubble Sort
algorithm

Step 3: Keep repeating for one fewer elements each time until there are no more pairs to compare.

unsorted

sorted

Bubble Sort
algorithm

Step 3: Keep repeating for one fewer elements each time until there are no more pairs to compare.

unsorted

sorted

Bubble Sort
algorithm

Step 3: Keep repeating for one fewer elements each time until there are no more pairs to compare.

unsorted

sorted

Bubble Sort

class exercise – write Bubble Sort

Homework!