

Computing Science 1P

COMPSCI 1001

Revision Lecture

March 13th, 2020

Lecturer: Dr Mohamed Khamis

Mohamed.Khamis@Glasgow.ac.uk

<https://www.gla.ac.uk/schools/computing/staff/mohamedkhamis/>

<http://mkhamis.com/>



University
of Glasgow

Announcements

- **Lab exam** has been replaced by an individual assessment. The assessment can now be downloaded from moodle and is due on Wednesday 25 March at 4:00pm.
 - Read the assessment specification and let me know if you have questions via the forum on Moodle, or via Sli.do. You can also email me.
- **Labs** are running next week (week commencing 16 March) but there are neither lab exercises nor attendance to be taken.
 - You can go during your lab slot to work on the assessed exercise. The tutors and demonstrators are there to answer generic questions (but not to discuss the assessment's solution).
 - My advice is to work from home and ask questions via moodle or sli.do.
 - Attendance: you are good if you attended
- There are **no labs** on the week commencing 23 March.
- Remember that all lectures are recorded! Including John's revision lecture from last term.

Questions?

- Post on sli.do #cs1p
- Post on the forum:
<https://moodle.gla.ac.uk/mod/forum/view.php?id=1467605>
- Or email me

Revision topics

- Files
- Sorting algorithms
- Searching
- Exception handling
- Graphical user interfaces

Sorting algorithms and complexity

Sli.do: Can you please go through all the sorting algorithms again?

Algorithm Analysis with Big-O Notation

- Constant Complexity ($O(1)$)
 - The number of operations required to execute an algorithm is constant.

```
def constant_algo(mylist):  
    result = mylist[0] * mylist[0]  
    print (result)  
constant_algo([4, 5, 6, 8])
```

- No matter how many elements are in mylist, this function will always perform two operations

Algorithm Analysis with Big-O Notation

- Linear Complexity ($O(n)$)
 - the number of operations required to complete the execution of an algorithm increase or decrease linearly with the number of inputs

```
def linear_algo(mylist):  
    for item in mylist:  
        print (item)  
linear_algo([4, 5, 6, 8])
```

- The number of iterations of the for-loop will be equal to the size of the input.

Algorithm Analysis with Big-O Notation

- Linear Complexity ($O(n)$)
 - describes an algorithm whose performance will grow linearly and in direct proportion to the size of the input data set

```
def linear_algo(mylist):  
    for item in mylist:  
        a = item * 2  
        print (a)  
linear_algo([4, 5, 6, 8])
```

- There are two operations per loop $\rightarrow 2n$ operations.
- Still, this is a linear function and its complexity is $O(n)$.

Algorithm Analysis with Big-O Notation

- Quadratic Complexity ($O(n^2)$)
 - represents an algorithm whose performance is directly proportional to the square of the size of the input data set

```
def linear_algo(mylist):  
    for item1 in mylist:  
        for item2 in mylist:  
            print (item1 * item2)  
linear_algo([4, 5, 6, 8])
```

- The number of operations depends on the size of two lists, i.e., $n \times n = n^2$

Algorithm Analysis with Big-O Notation

- Quadratic Complexity ($O(n^2)$)
 - represents an algorithm whose performance is directly proportional to the square of the size of the input data set

```
def linear_algo(mylist1, mylist2):  
    for item1 in mylist1:  
        for item2 in mylist2  
            print (item1 * item2)  
linear_algo(4, 5, 6, 8], [1,2,3])
```

- Here we have two lists of different sizes $\rightarrow m \times n$
- Still, this is a quadratic function and its complexity is represented as $O(n^2)$.

Algorithm Analysis with Big-O Notation

- Exponential Complexity ($O(2^n)$)
 - represents an algorithm whose runtime is doubled with each addition to the input data set.

```
def fibonacci(number):  
    if (number <= 1) return number  
    return fibonacci(number - 2) + fibonacci(number - 1)
```

- Each call to fibonacci makes two calls to fibonacci.
- Examples:
 - fibonacci(1) makes $2^0 = 1$ calls to fibonacci()
 - fibonacci(2) makes $2^1 = 2$ calls to fibonacci()
 - fibonacci(3) makes $2^2 = 4$ calls to fibonacci()
 - ...

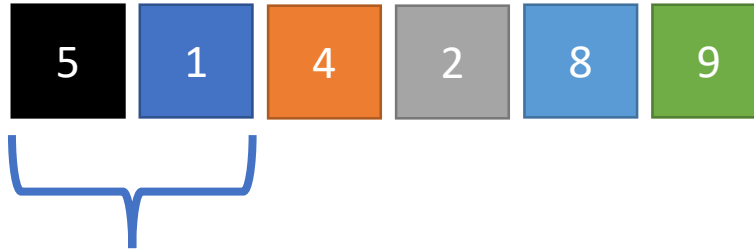
Algorithm Analysis with Big-O Notation

- You can implement things differently to reduce complexity and improve performance!
- For example, using dictionaries with fibonacci reduces it from $O(2^n)$ to $O(n)$.

Bubble sort

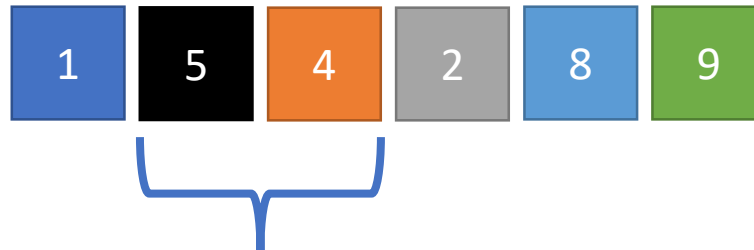
- Algorithm:
 - Compare two neighboring elements.
 - Swap them if they are in the wrong order.
 - Repeat until the list is fully sorted.

Bubble sort example



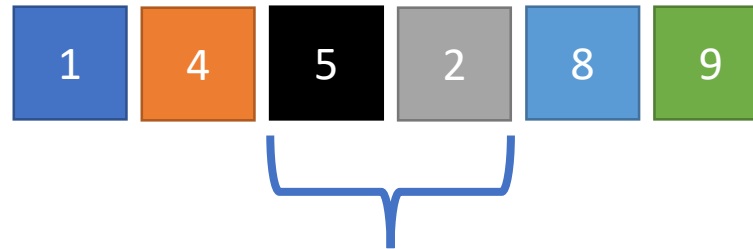
If $5 > 1$:
swap(5, 1)

Bubble sort example



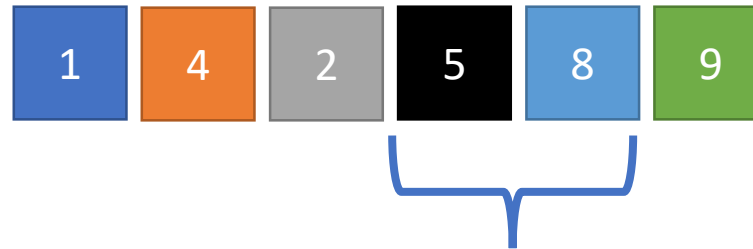
If $5 > 4$:
swap(5, 4)

Bubble sort example



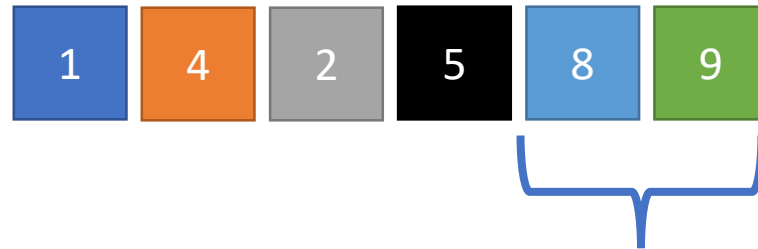
If $5 > 2$:
swap (5, 2)

Bubble sort example



If $5 > 8$:
swap (5 , 8)

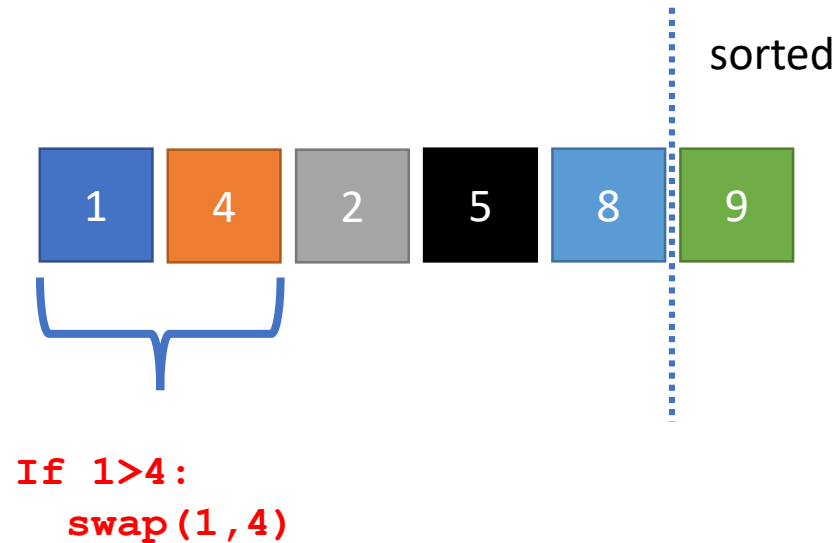
Bubble sort example



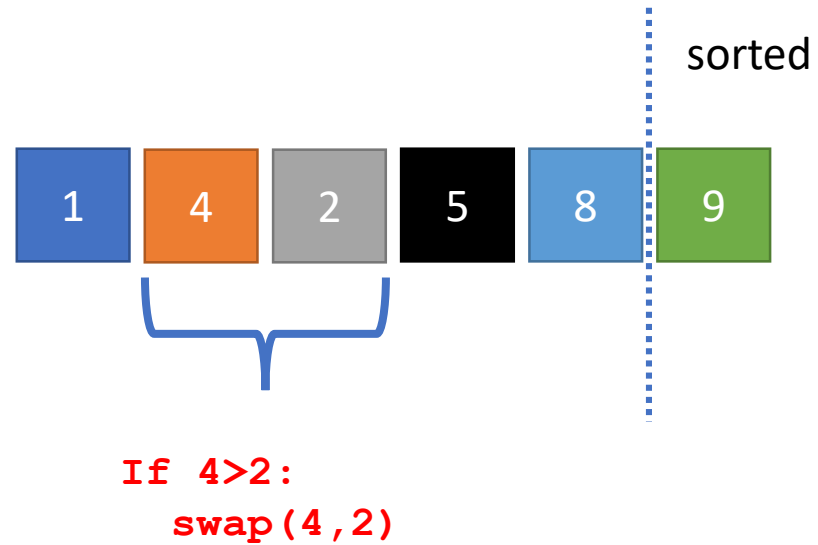
If $8 > 9$:
swap (8 , 9)

After one pass, the last element is in the correct position now!

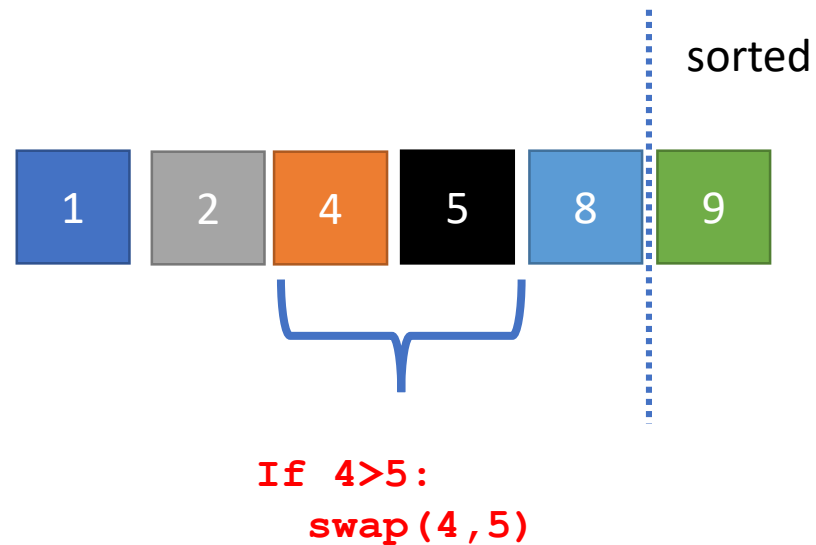
Bubble sort example



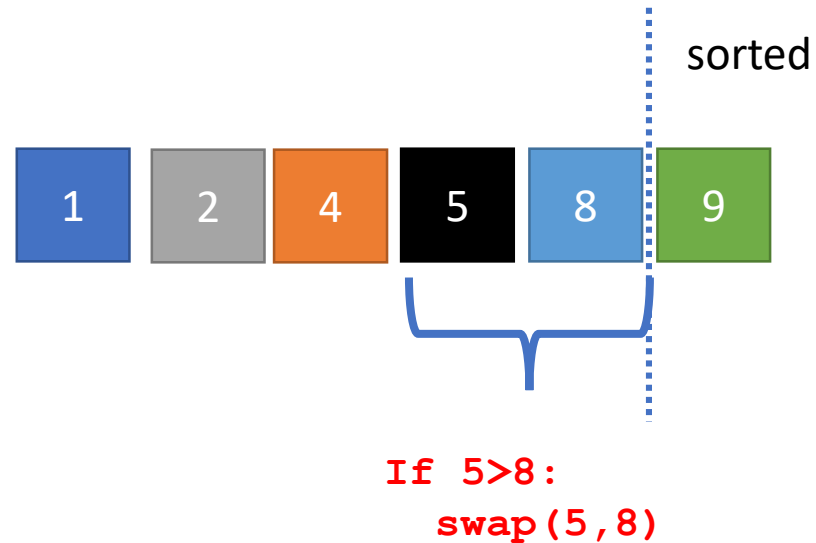
Bubble sort example



Bubble sort example



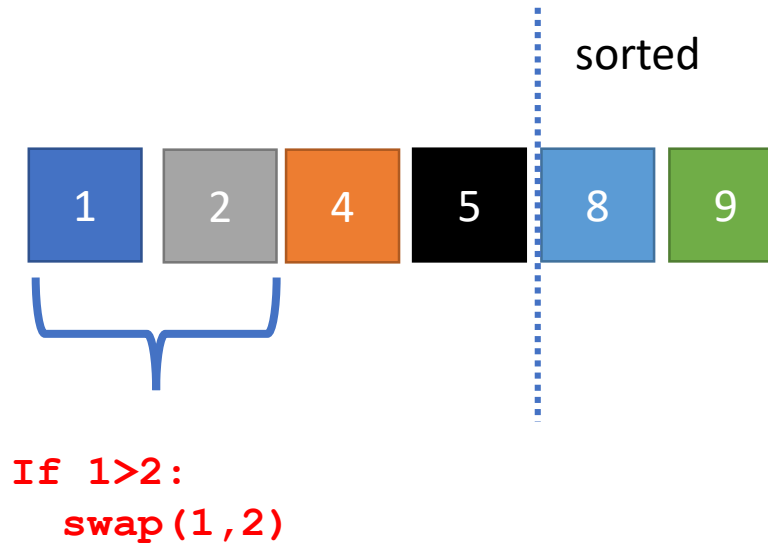
Bubble sort example



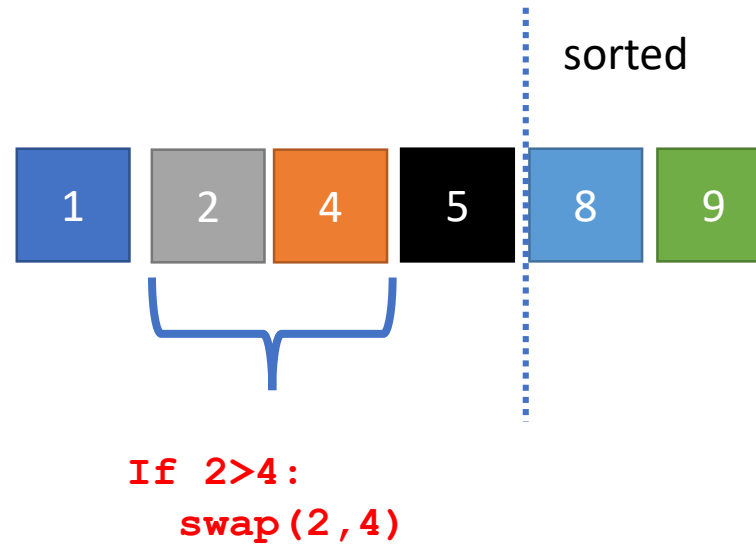
We don't need to check the last pair – we know that these are sorted already!

Now that the second pass is over, the last two digits are in the right place. Let's sort the rest.

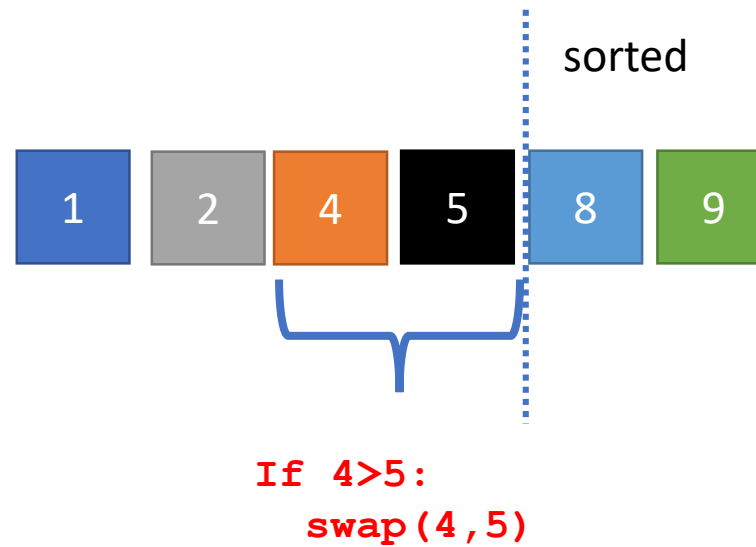
Bubble sort example



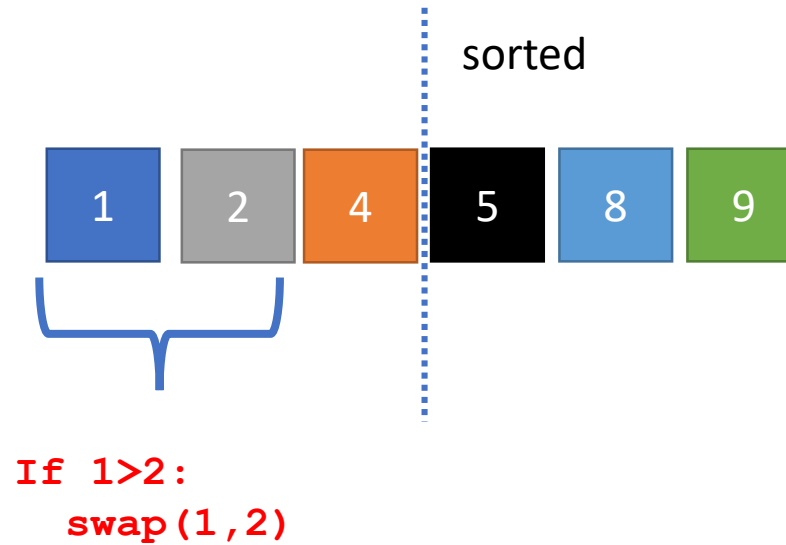
Bubble sort example



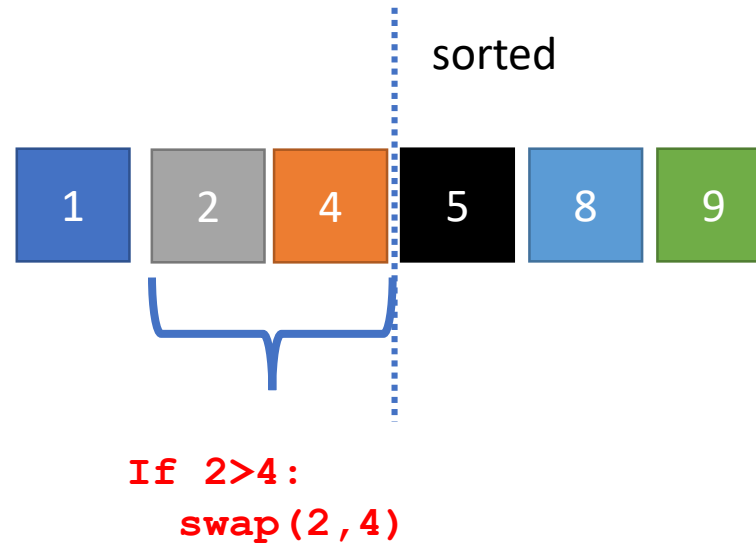
Bubble sort example



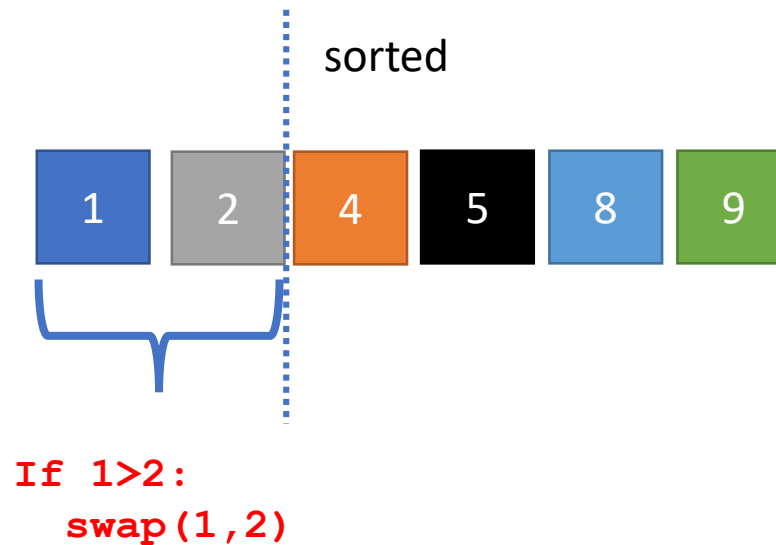
Bubble sort example



Bubble sort example



Bubble sort example



Bubble sort

- Algorithm:
 - Compare two neighboring elements.
 - Swap them if they are in the wrong order.
 - Repeat until the list is fully sorted.
- Question:
 - For a list of size n , How many passes do you need until the list is fully sorted?
 - Use bubble sort to sort $[5,4,3,2,1]$. How many passes do you need?

Bubble sort in Python

```
def bubbleSort(alist):  
    for position, item in enumerate(alist):  
        for i in range(len(alist)-1 - position):  
            if alist[i]>alist[i+1]: # compare the two neighbors  
                alist[i], alist[i+1] = alist[i+1], alist[i] # swap  
  
alist = [54,26,93,17,77,31,44,55,20]  
bubbleSort(alist)  
print(alist)
```

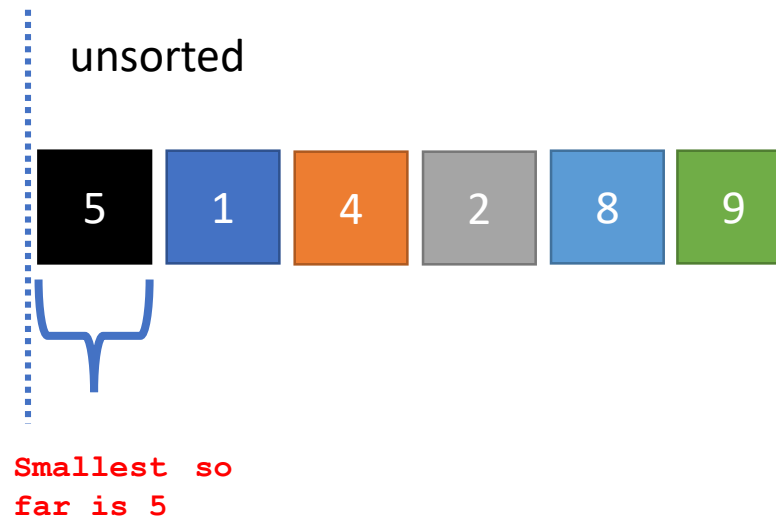
Efficiency of the Bubble Sort algorithm

- N is the number of items in the array.
- There are $n - 1$ comparisons on the first pass.
- There are $n - 2$ comparisons on the second pass, and so on.
- Total number of comparisons: $(n - 1) + (n - 2) + (n - 3) + \dots + 3 + 2 + 1$
- This is the same as $\frac{n(n-1)}{2}$
- In worst case, the list is inversely sorted and thus a swap is necessary in every comparison. This results in $n \times n$ swaps $\rightarrow O(n^2)$
- In best case, the list is already sorted so you don't do any swaps. You can optimize the algorithm so it would be $O(n)$.

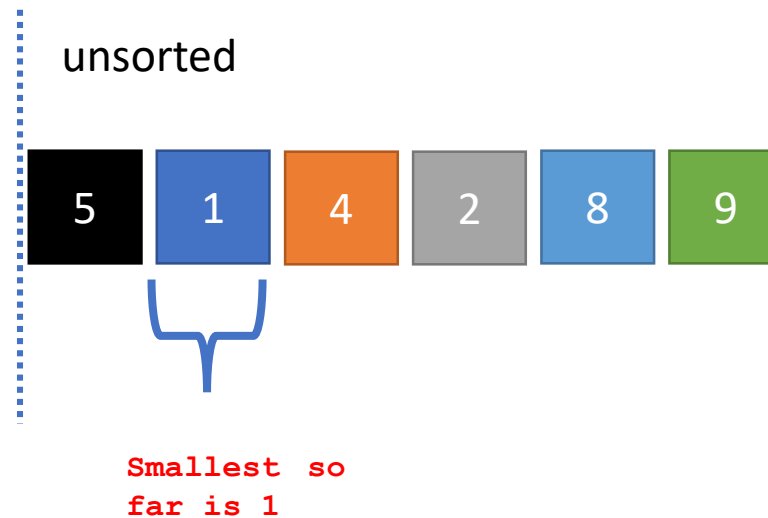
Selection sort

- Algorithm
 - Find the smallest element in a list, and swap it with the first element
 - Find the second smallest element in the list, and swap it with the second element
 - Continue until last element is reached
- It is called selection sort because every time the smallest remaining element is selected.

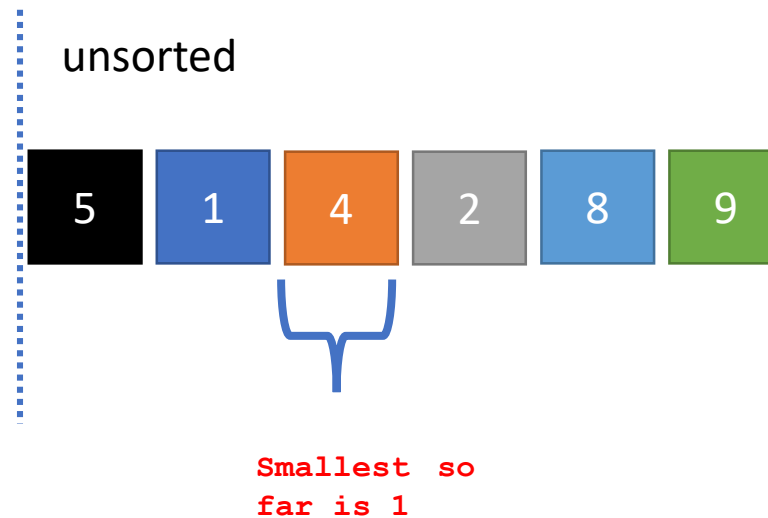
Selection sort example



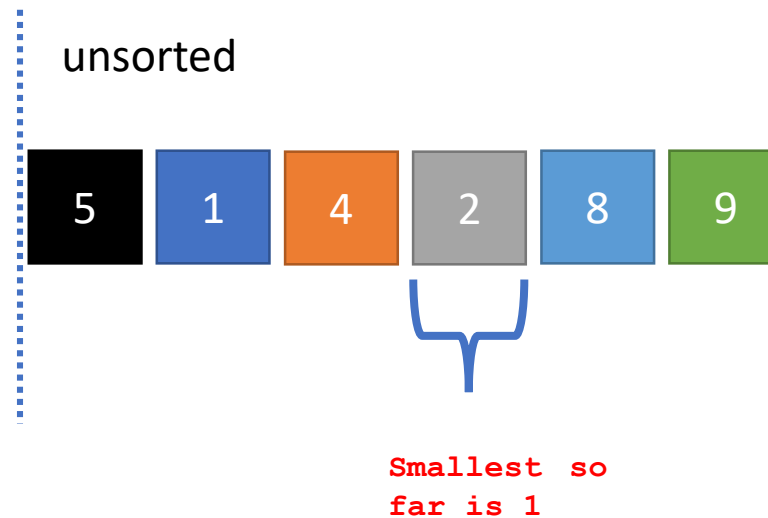
Selection sort example



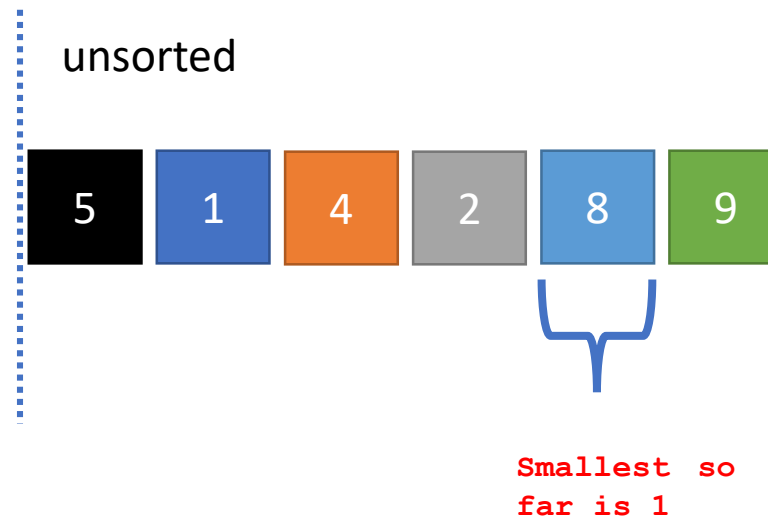
Selection sort example



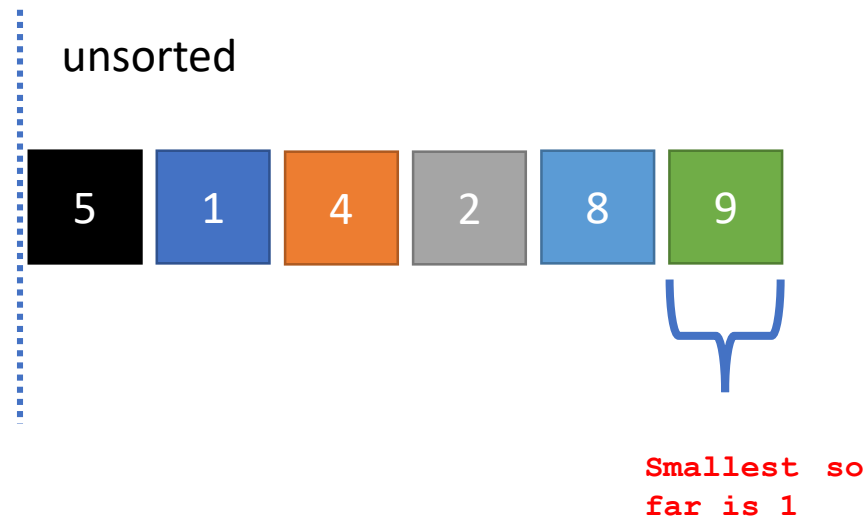
Selection sort example



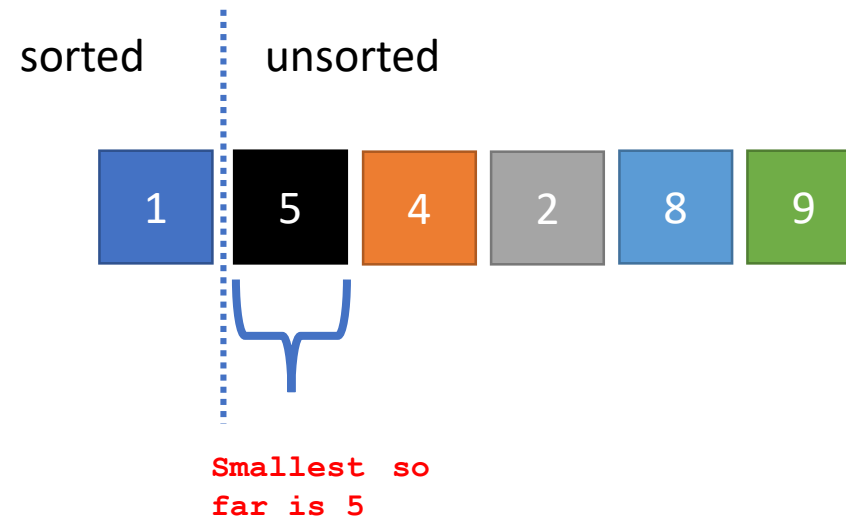
Selection sort example



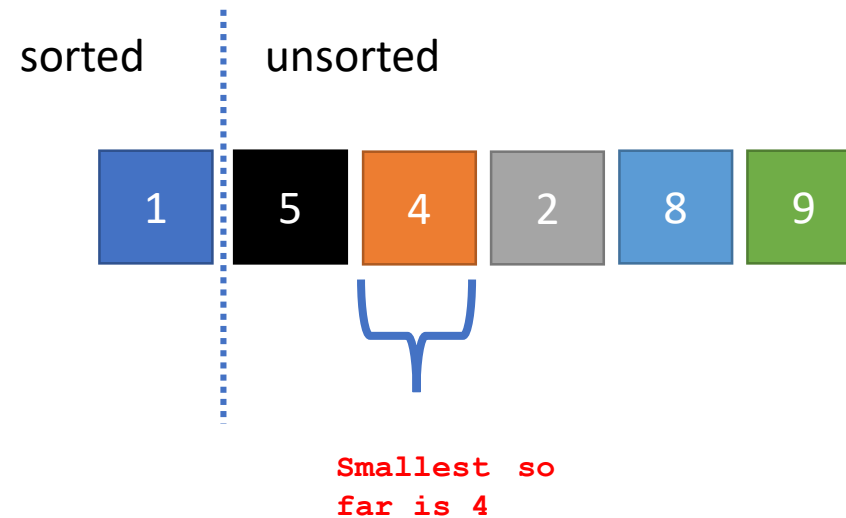
Selection sort example



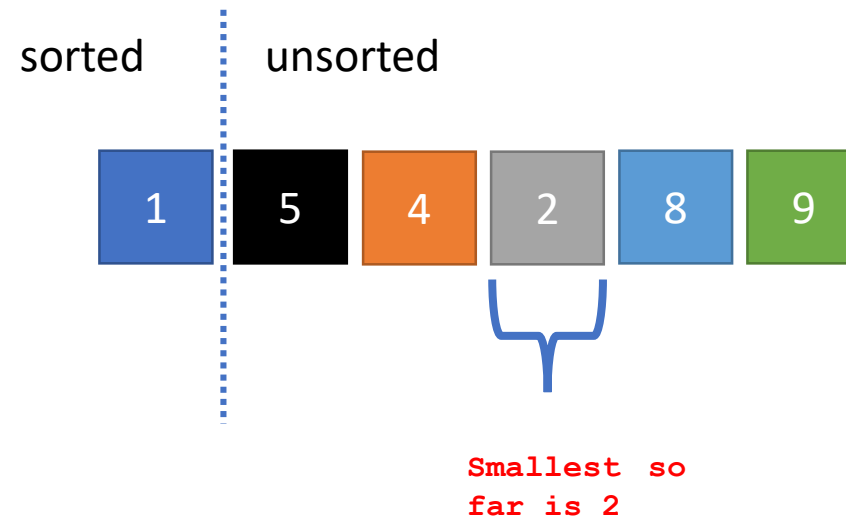
Selection sort example



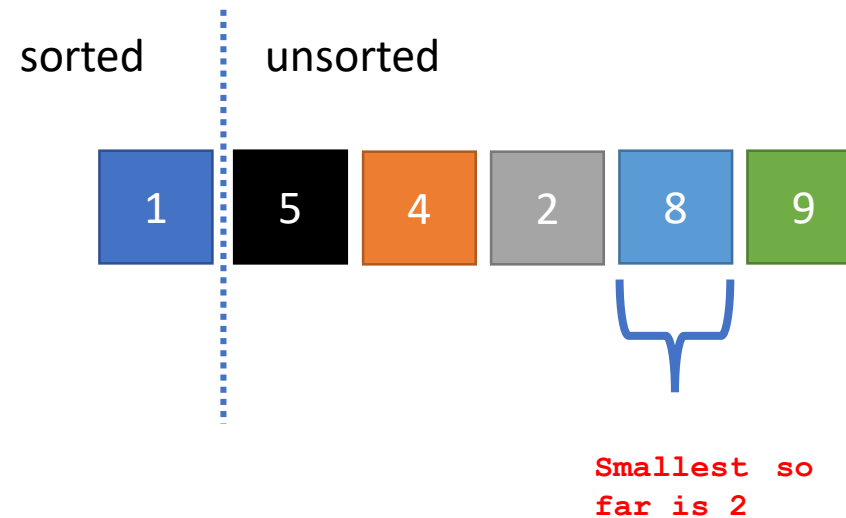
Selection sort example



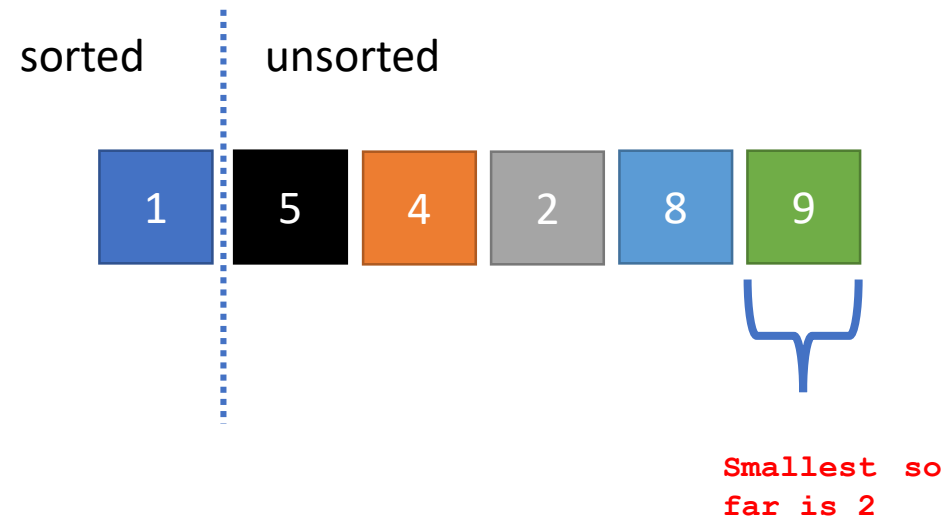
Selection sort example



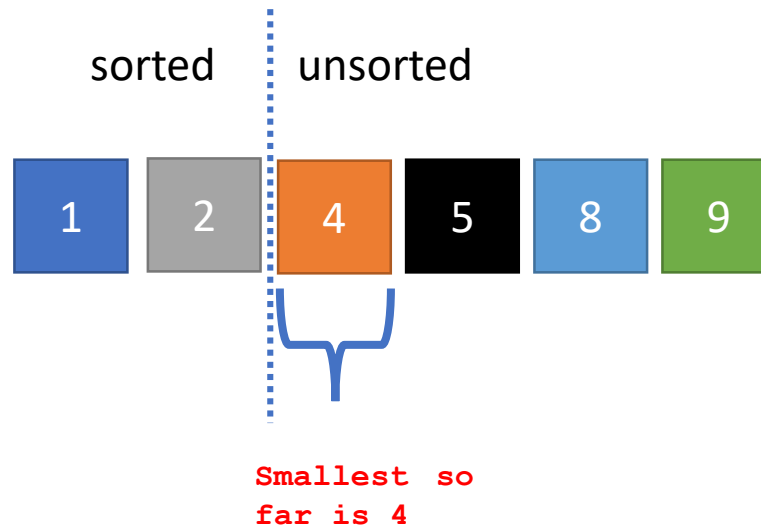
Selection sort example



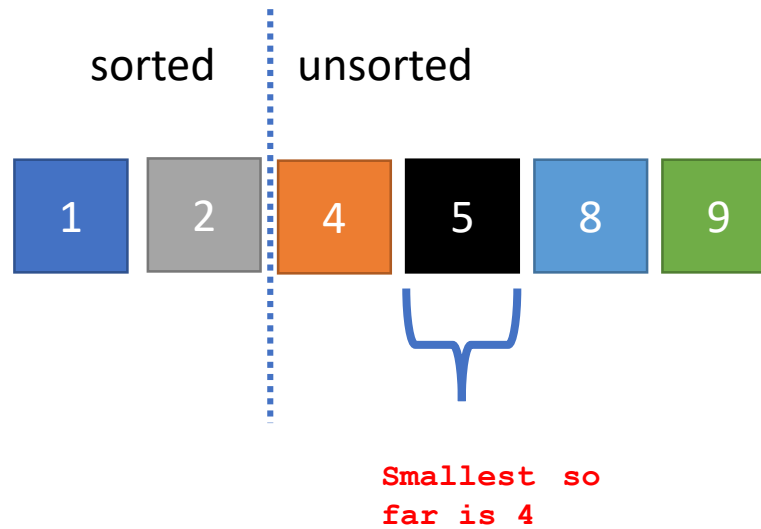
Selection sort example



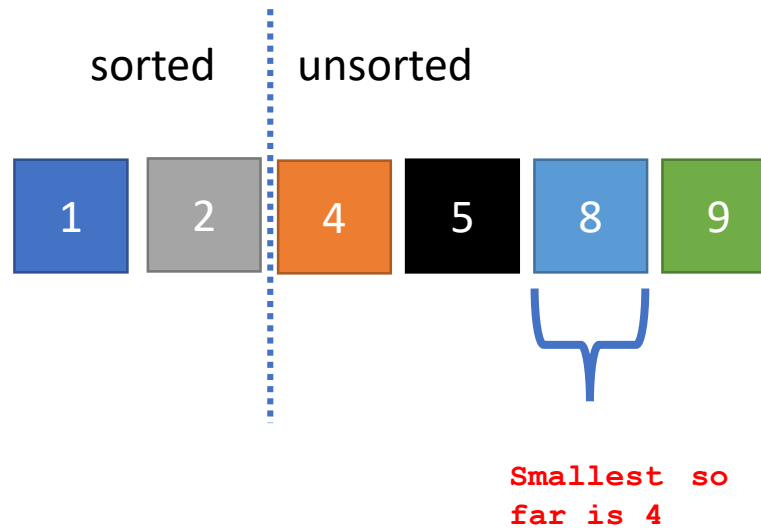
Selection sort example



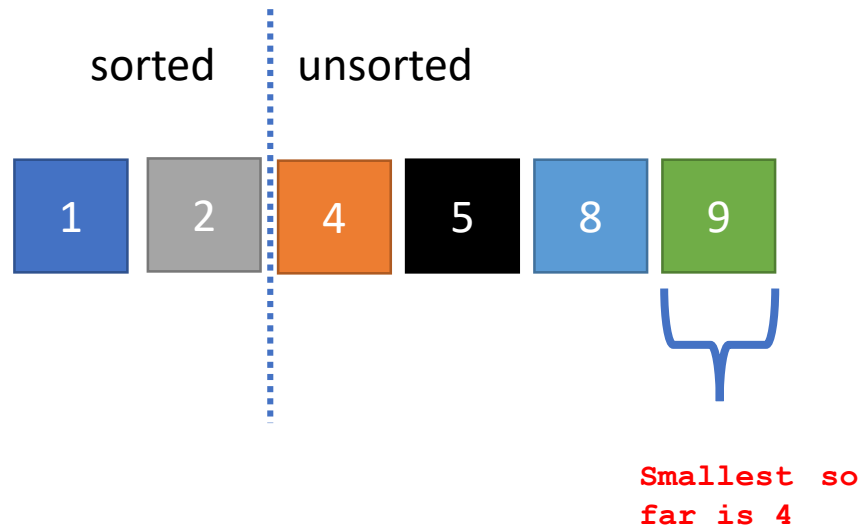
Selection sort example



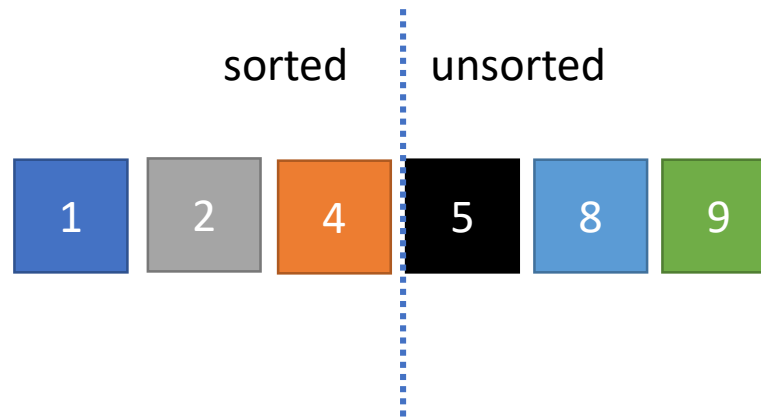
Selection sort example



Selection sort example



Selection sort example



And so on..

Selection sort in Python

```
def selectionSort(alist):  
    for i in range(len(alist)):  
        min_so_far = i  
        # find the minimum so far in the unsorted part of the list  
        for j in range(i+1, len(alist)):  
            if alist[min_so_far] > alist[j]:  
                min_so_far = j  
        # Swap the found minimum element with  
        # the first element of the unsorted list  
        alist[i], alist[min_so_far] = alist[min_so_far], alist[i]  
    return alist  
  
alist = [54,26,93,17,77,31,44,55,20]  
selectionSort(alist)  
print(alist)
```

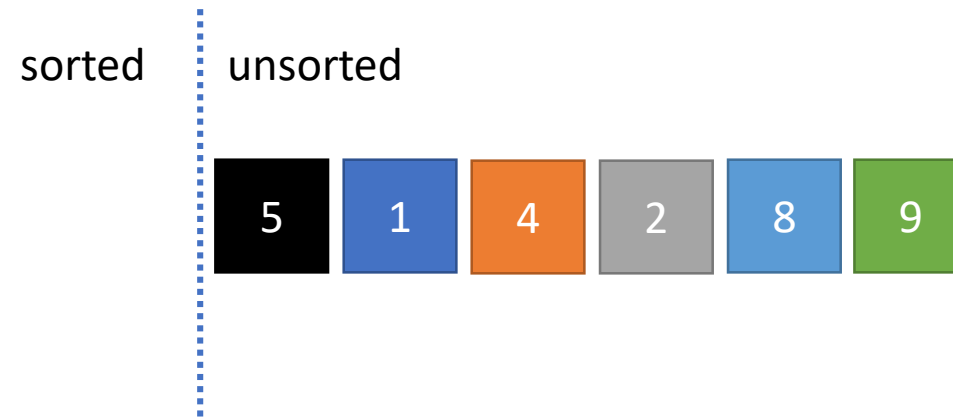
Efficiency of the Selection Sort algorithm

- N is the number of items in the array.
- There are $n - 1$ comparisons on the first pass.
- There are $n - 2$ comparisons on the second pass, and so on.
- Total number of comparisons: $(n - 1) + (n - 2) + (n - 3) + \dots + 3 + 2 + 1$
- This is the same as $\frac{n(n-1)}{2}$
- In worst case, the list is inversely sorted and thus a swap is necessary in every comparison. This results in $n \times n$ swaps $\rightarrow O(n^2)$
- Unlike the bubble sort, even if the list is sorted you still need to search for the minimum and put it in the first position (or at least make sure it is in the first position) So best-case time-complexity is still $O(n^2)$.

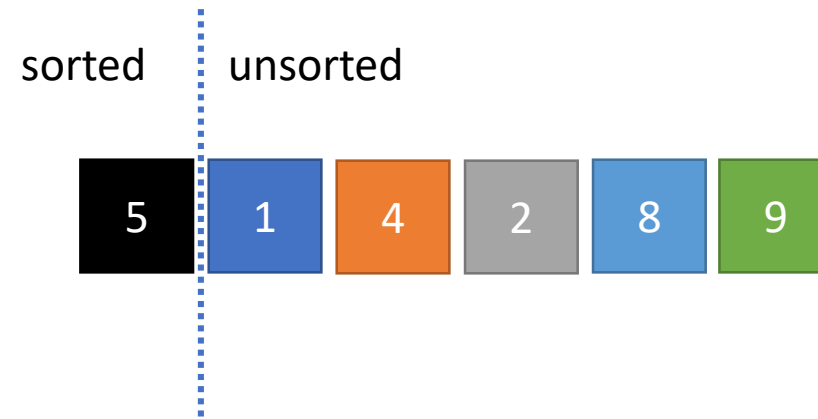
Insertion sort

- Algorithm
 - Start from left to right
 - Examine each element and compare it to the elements on its left.
 - Insert the current element in the correct position in the sorted part so far. The part of the list that is on the left of the current element is the sorted one so far.
- It is similar to the common way of ordering cards.

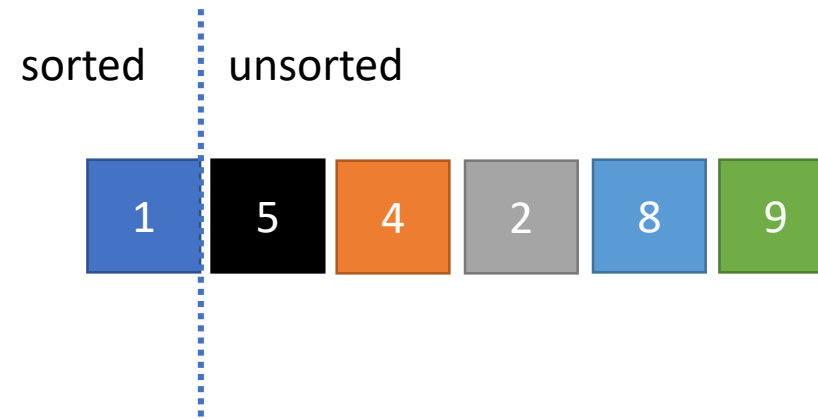
Insertion sort



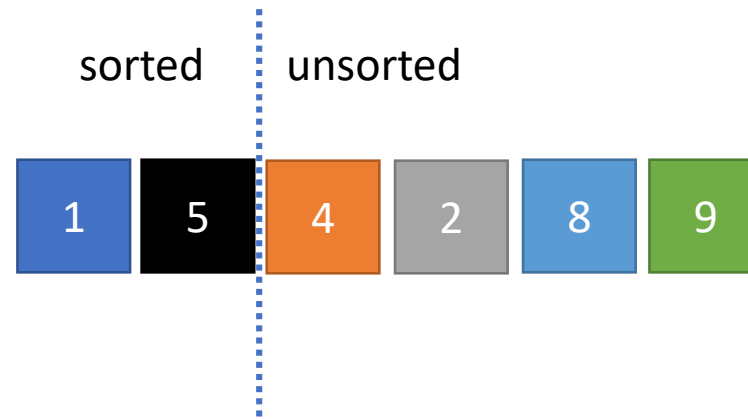
Insertion sort



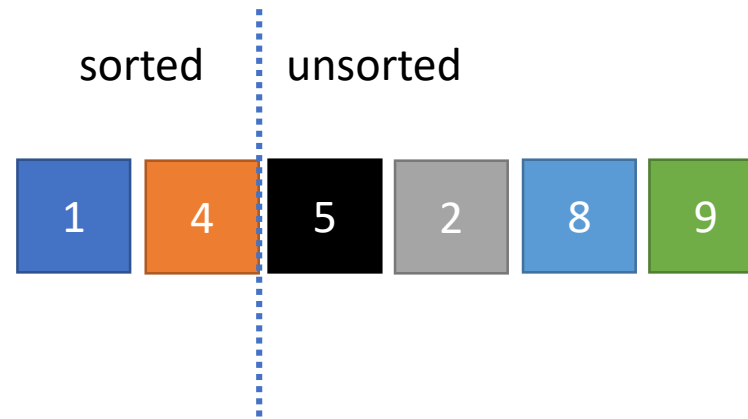
Insertion sort



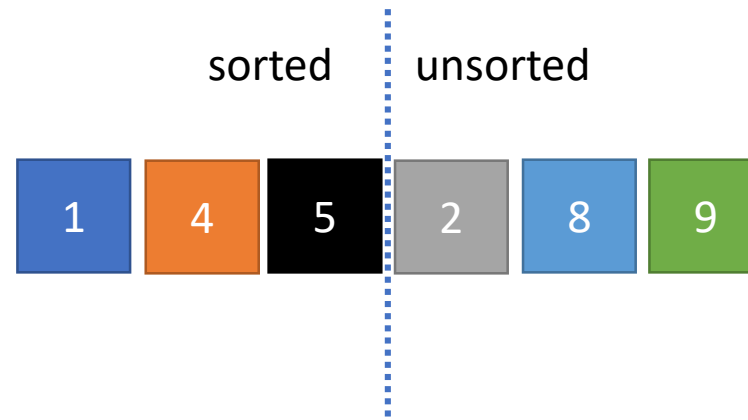
Insertion sort



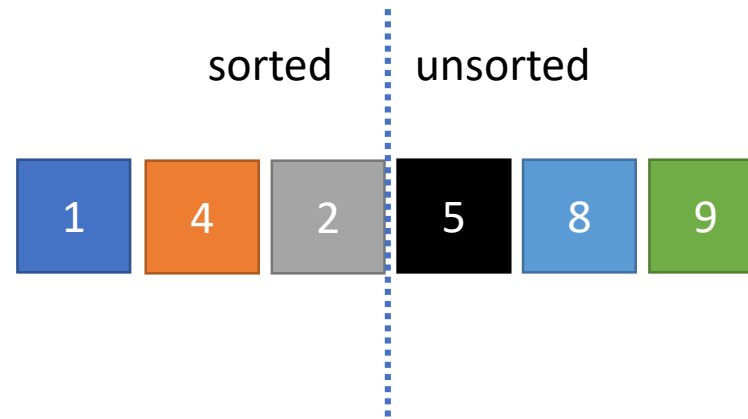
Insertion sort



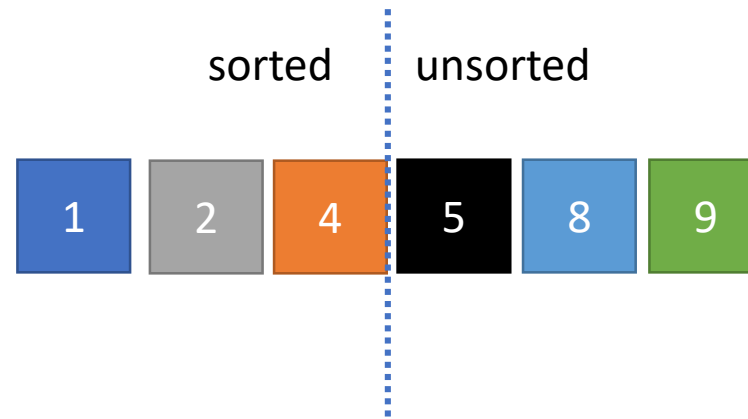
Insertion sort



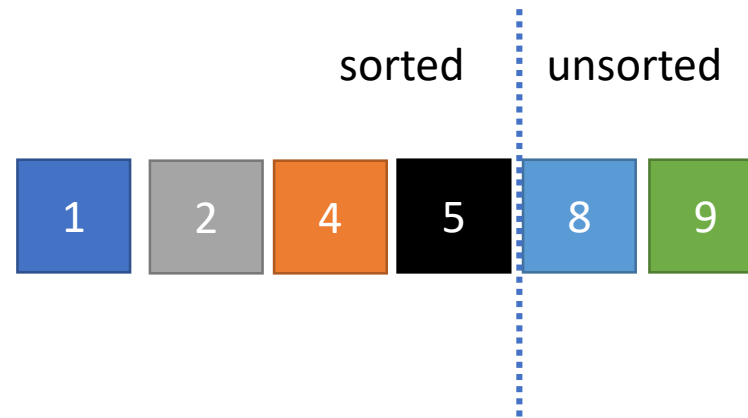
Insertion sort



Insertion sort



Insertion sort



And so on..

Insertion sort in python

```
def insertionSort(alist):
    for index in range(1,len(alist)):
        currentvalue = alist[index]
        position = index
        # Move elements of alist[0..index], that are
        # greater than currentvalue one position ahead
        while position>0 and alist[position-1]>currentvalue:
            alist[position] = alist[position-1]
            position = position-1
        # now all the elements that are greater than currentvalue are on the right of
        # position. So we can put the current value in position.
        alist[position]=currentvalue

alist = [12,11,13,5,6]
insertionSort(alist)
print(alist)
```

Efficiency of the Insertion Sort algorithm

- N is the number of items in the array.
- $2 \times (1 + 2 + \dots + n - 2 + n - 1)$
- There are $n - 1$ comparisons on the first pass.
- There are $n - 2$ comparisons on the second pass, and so on.
- Total number of comparisons: $(n - 1) + (n - 2) + (n - 3) + \dots + 3 + 2 + 1$
- This is the same as $\frac{n(n-1)}{2}$
- In worst case, the list is inversely sorted and thus a swap is necessary in every comparison. This results in $n \times n$ swaps $\rightarrow O(n^2)$
- In best case, the list is already sorted so you don't do any swaps, but you still need to go through it once to check $\rightarrow O(n)$

Invariants

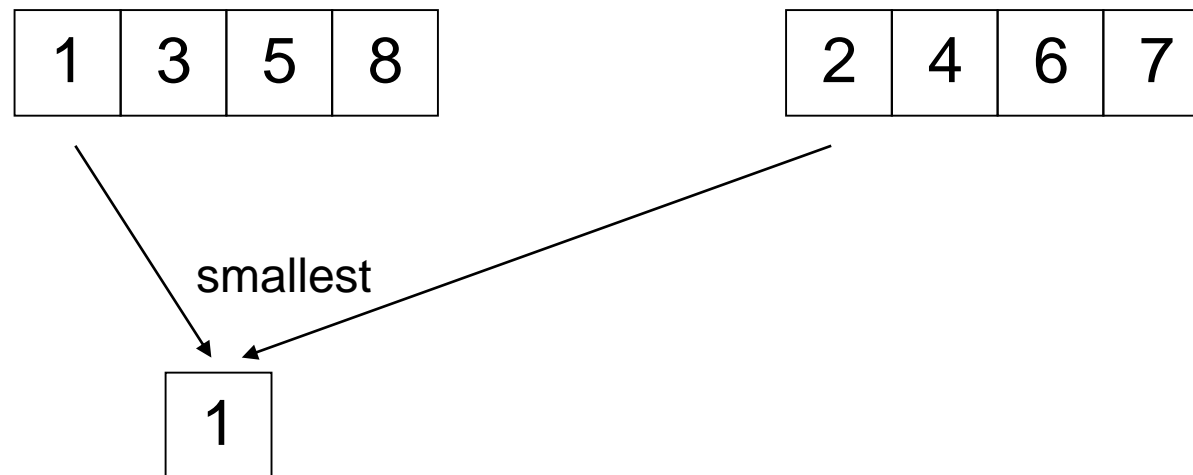
- Invariant: a statement that remains unchanged as the algorithm proceeds.
- Bubble sort: the invariant is that the last i list elements are sorted.
- Selection sort: the invariant is that the list elements with indices less than or equal to i are sorted.
- Insertion sort: the invariant is that the list elements with indices less than or equal to i are sorted.

Summary so far

- Bubble sort
 - Compare each pair and sorts them
 - Sort the pair.
 - Repeat.
 - At every iteration n , the last $n-1$ elements are sorted.
 - Time-Complexity: $O(n^2)$
- Selection sort
 - Go through the list to find the minimum.
 - Put it at the beginning of the unsorted part of the list.
 - At every iteration n , the first $n-1$ elements are sorted.
 - Time-Complexity: $O(n^2)$
- Insertion sort
 - Go through the list
 - Place every element you encounter in its correct position of the sorted part of the list
 - At every iteration n , the first $n-1$ elements are sorted.
 - Time-Complexity: $O(n^2)$

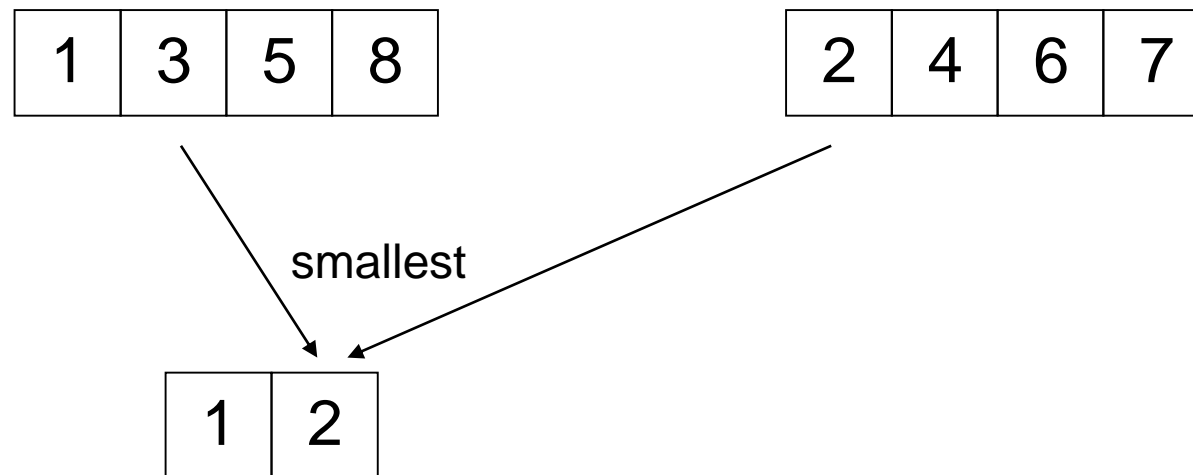
Merge sort

- Merging two sorted lists:



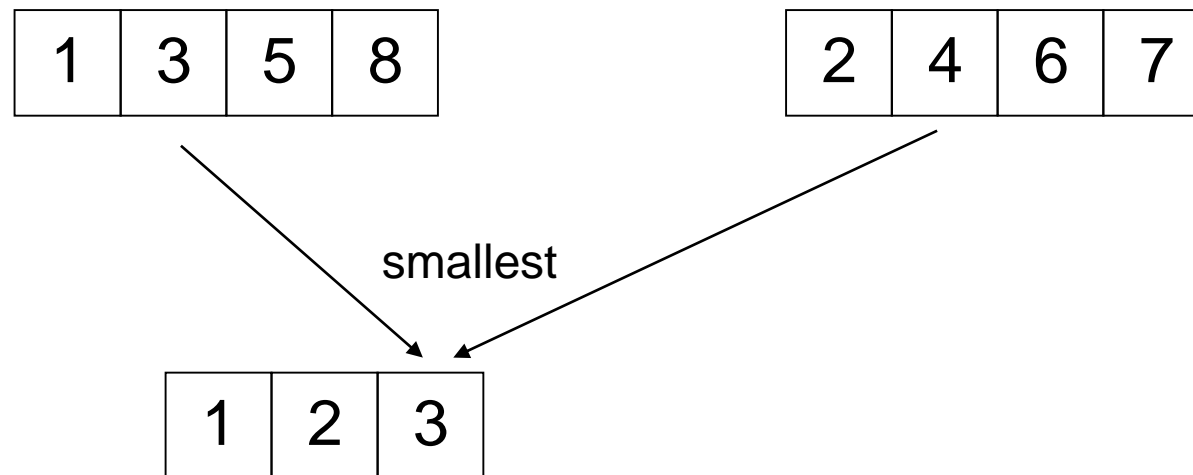
Merge sort

- Merging two sorted lists:



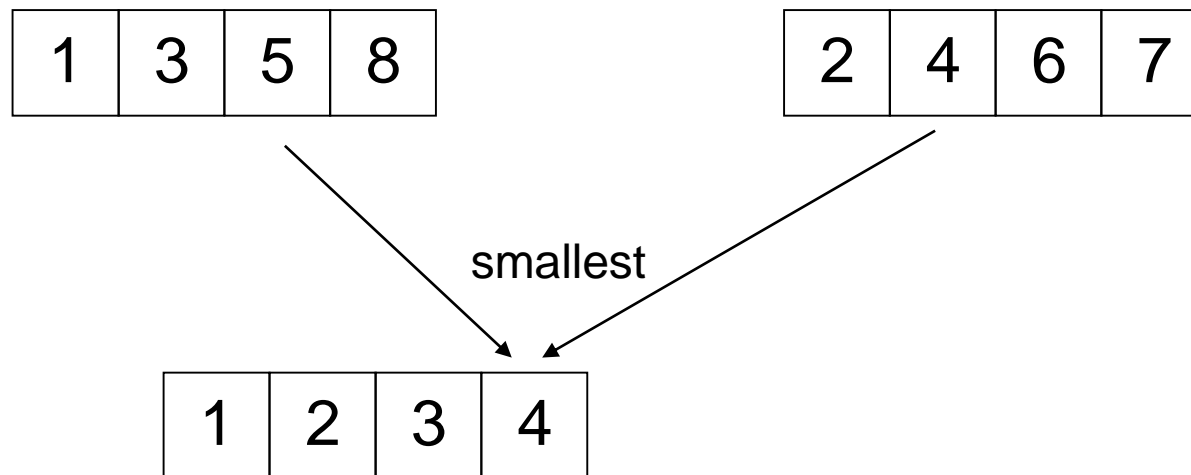
Merge sort

- Merging two sorted lists:



Merge sort

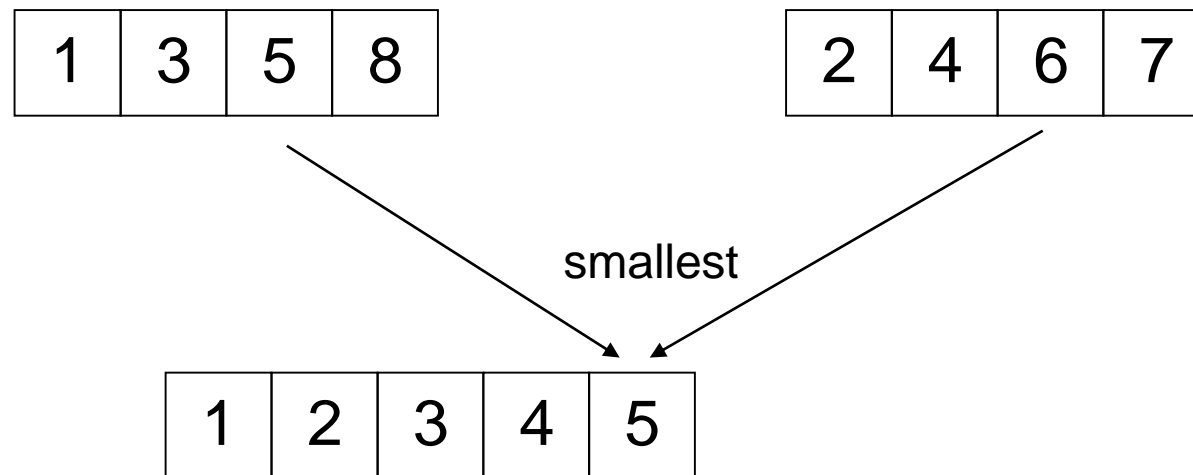
- Merging two sorted lists:



- Notice at each stage you are just comparing two items in each list

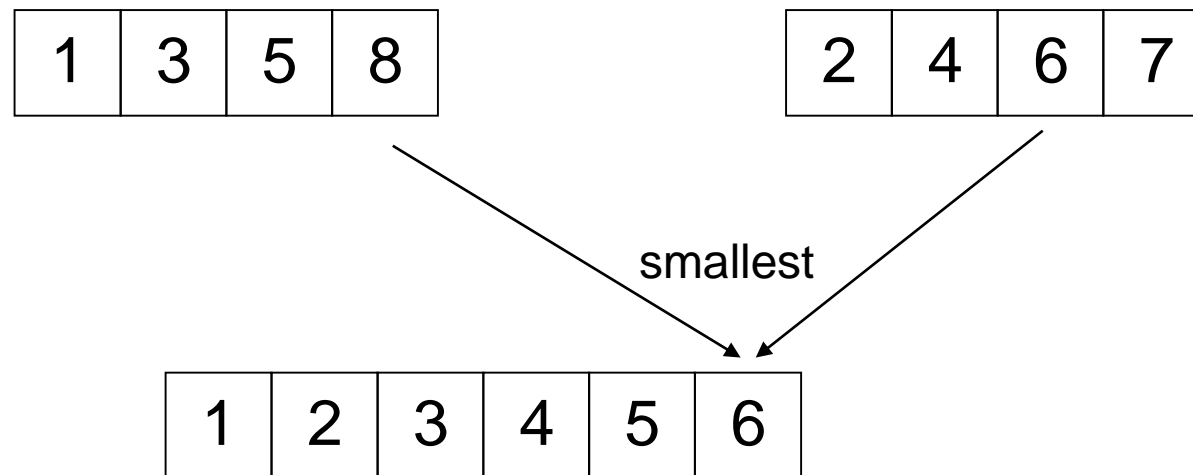
Merge sort

- Merging two sorted lists:



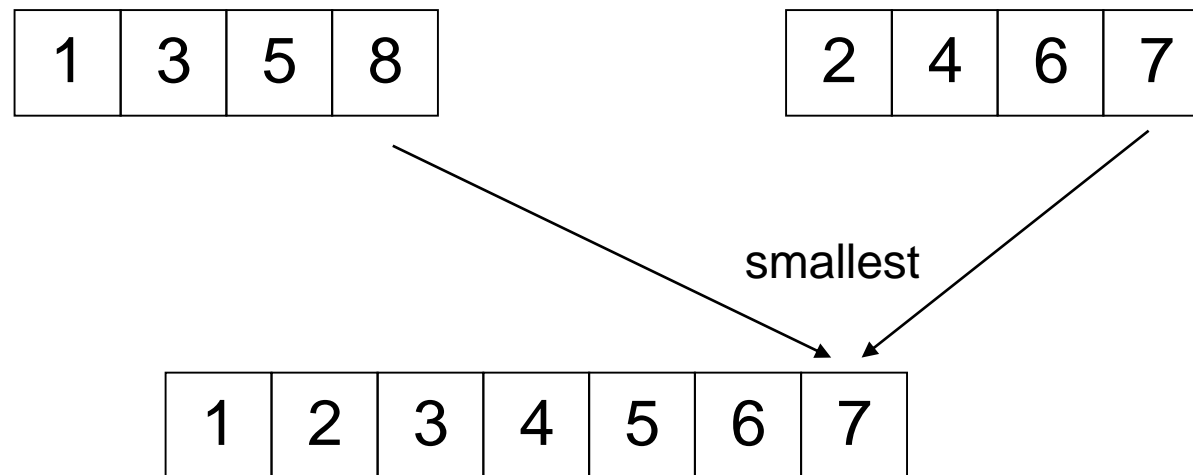
Merge sort

- Merging two sorted lists:



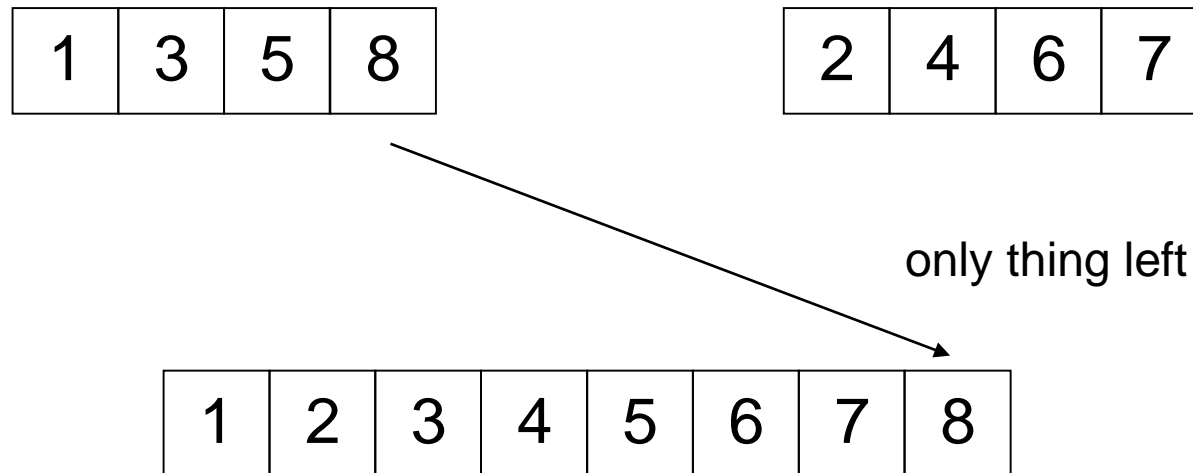
Merge sort

- Merging two sorted lists:



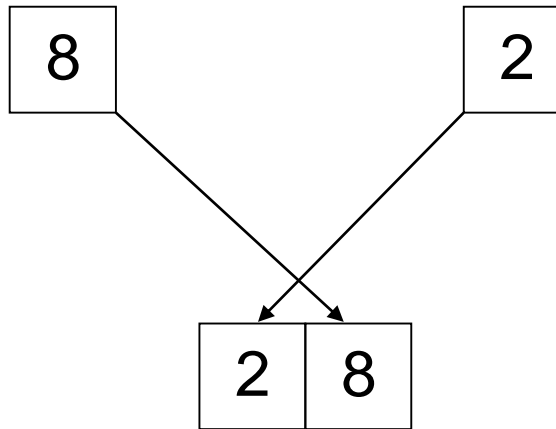
Merge sort

- Merging two sorted lists:

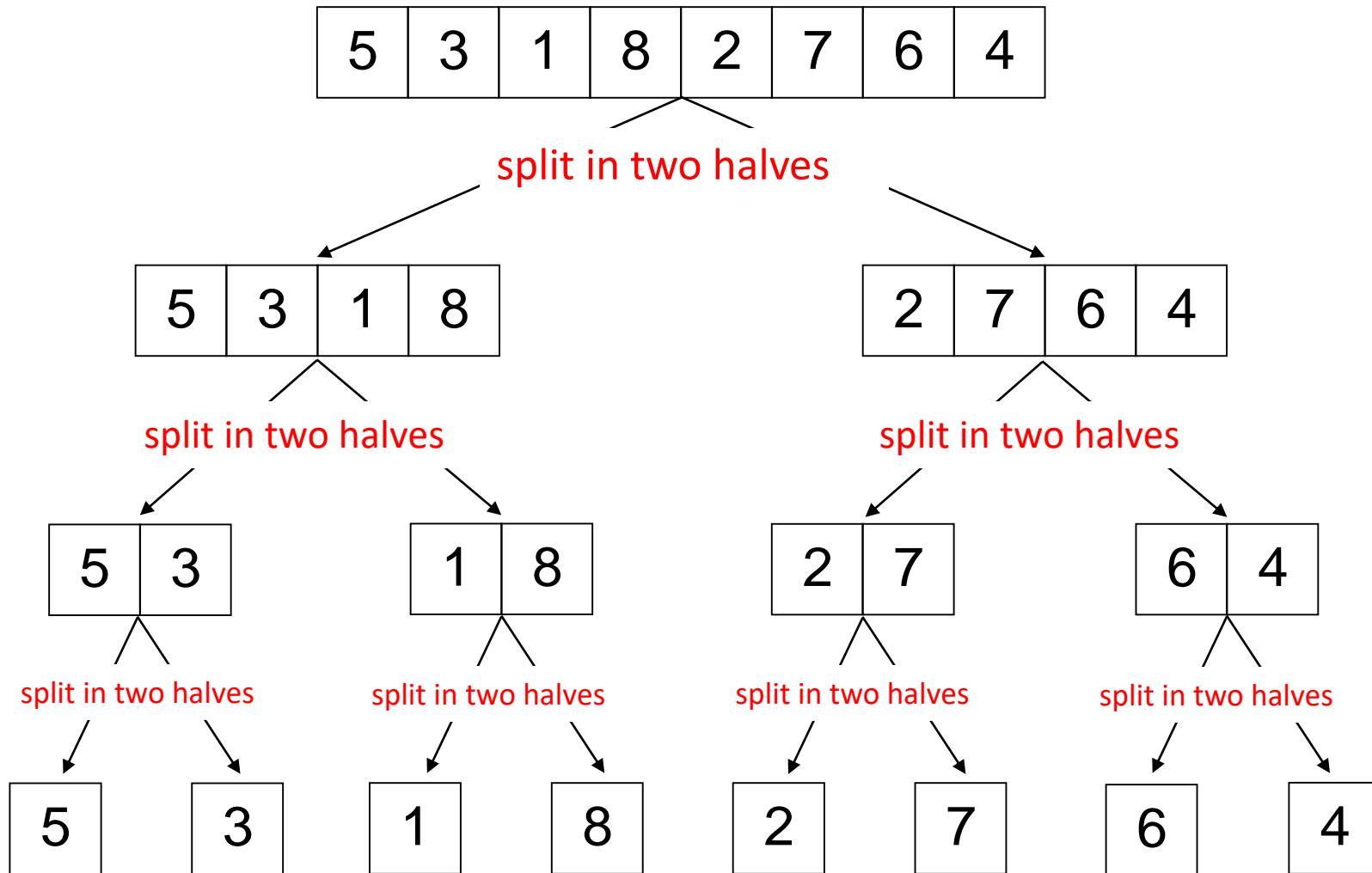


Merge sort

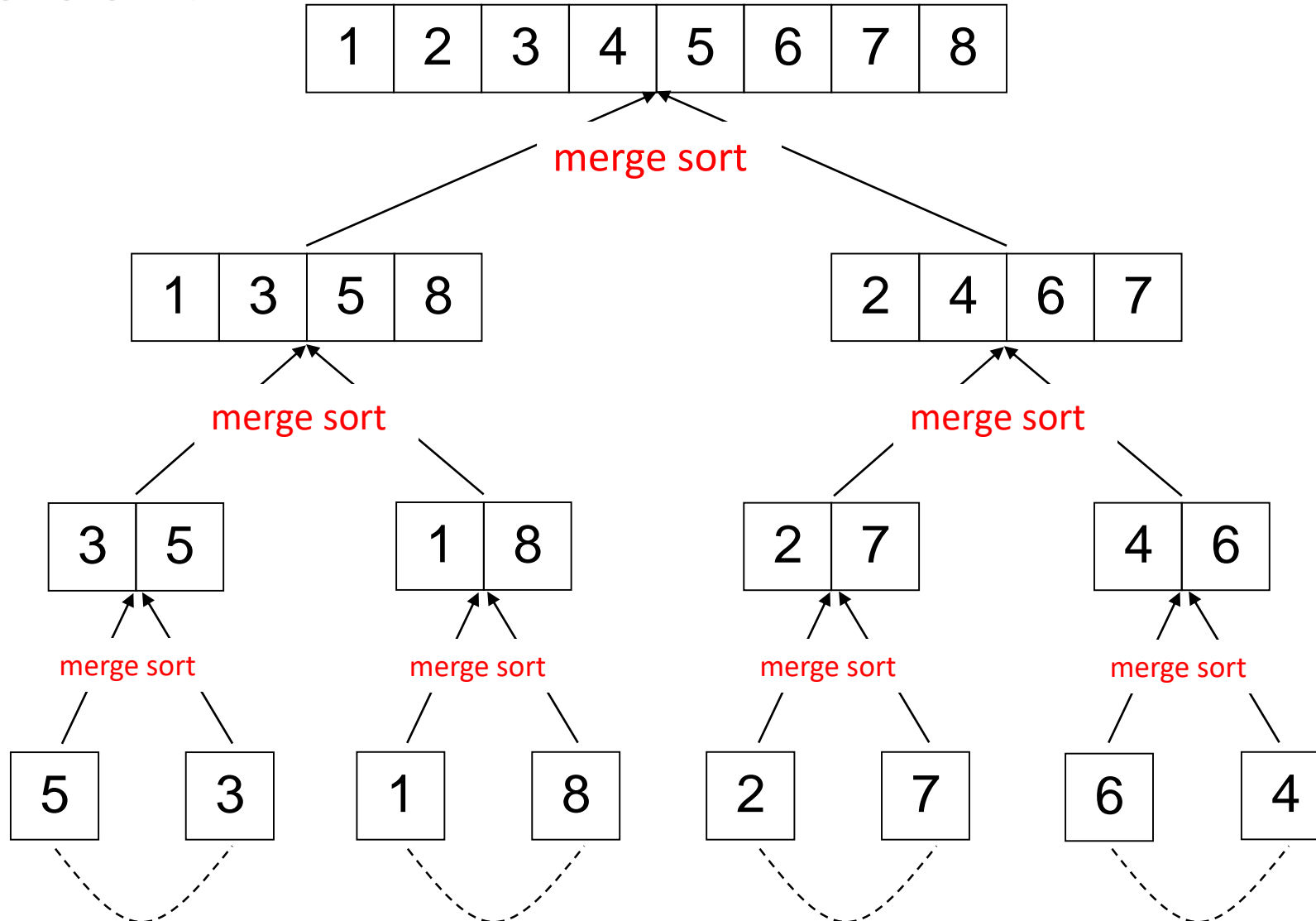
- Let's call that function `merge()`
- What would `merge()` do when given two lists, each consisting of one element?



Merge sort



Merge sort



Merge sort in Python

```
def merge(x,y):
    i = 0    # position in x
    j = 0    # position in y
    z = []   # new list
    while i < len(x) and j < len(y):
        if x[i] < y[j]: # next item comes from x
            z = z + [x[i]]
            i = i + 1
        else: # next item comes from y
            z = z + [y[j]]
            j = j + 1
    if i < len(x): # unmerged items are remaining in x, add them to z
        z = z + x[i:]
    else: # unmerged items are remaining in y, add them to z
        z = z + y[j:]
    return z
```

Merge sort in Python

```
def sort(x):  
    if len(x) <= 1:  
        return x  
    else:  
        d = len(x)//2  
        return merge(sort(x[:d]), sort(x[d:]))
```

Python3:

```
>>> 10 / 3  
3.3333333333333335
```

and in Python 2.x:

```
>>> 10 / 3  
3
```

Floor division

`merge(sort(x[:len(x)//4]), sort(x[len(x)//4:len(x)//2]))`

`merge(sort(x[len(x)//2:3*len(x)//4]), sort(x[3*len(x)//4:]))`

Analyzing the Merge sort algorithm

- There are $\log n$ rounds of merging, each requiring n comparisons
- We say that merge sort has $\text{order } n \log n$
- Invariant: at the start of each inner iteration k in the merge function, the nonempty part of z contains the $k - 1$ smallest elements of x and y , in sorted order. Moreover, $x[i]$ and $y[j]$ are the smallest elements of their arrays that have not been copied to z

Quick sort

- Another popular algorithm that is also order of $n(\log n)$
- Core idea:
 - Pick an element: we'll call it the pivot
 - Move all elements smaller than the pivot to its left
 - Move all elements larger than the pivot to its right



Quick sort

- Another popular algorithm that is also order of $n(\log n)$
- Core idea:
 - Pick an element: we'll call it the pivot
 - Move all elements smaller than the pivot to its left
 - Move all elements larger than the pivot to its right



1. Swap your pivot with the element at the end of the list

Quick sort

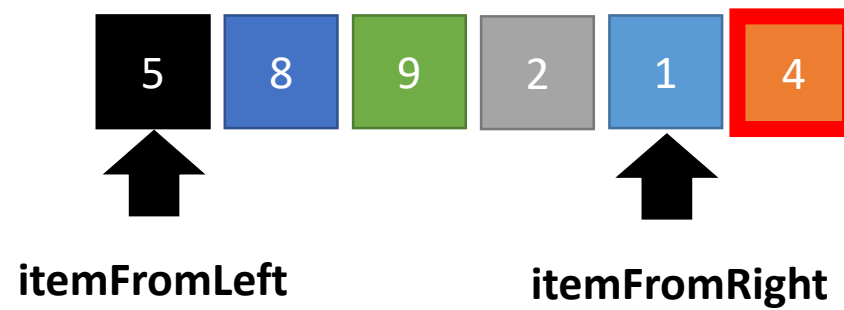
- Another popular algorithm that is also order of $n(\log n)$
- Core idea:
 - Pick an element: we'll call it the pivot
 - Move all elements smaller than the pivot to its left
 - Move all elements larger than the pivot to its right



2. Look for 2 items:
 - **itemFromLeft** that is larger than the pivot
 - **itemFromRight** that is smaller than the pivot

Quick sort

- Another popular algorithm that is also order of $n(\log n)$
- Core idea:
 - Pick an element: we'll call it the pivot
 - Move all elements smaller than the pivot to its left
 - Move all elements larger than the pivot to its right

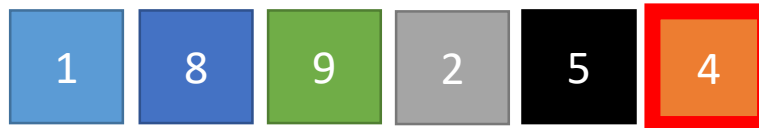


2. Look for 2 items:
 - **itemFromLeft** that is larger than the pivot
 - **itemFromRight** that is smaller than the pivot

3. Swap them!

Quick sort

- Another popular algorithm that is also order of $n(\log n)$
- Core idea:
 - Pick an element: we'll call it the pivot
 - Move all elements smaller than the pivot to its left
 - Move all elements larger than the pivot to its right

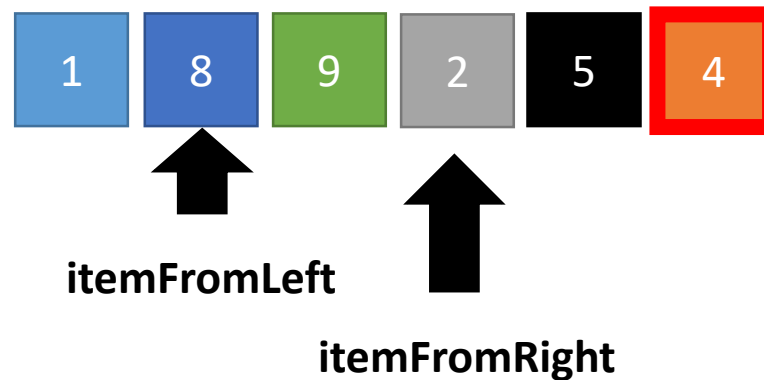


2. Look for 2 items:
 - **itemFromLeft** that is larger than the pivot
 - **itemFromRight** that is smaller than the pivot

3. Swap them!

Quick sort

- Another popular algorithm that is also order of $n(\log n)$
- Core idea:
 - Pick an element: we'll call it the pivot
 - Move all elements smaller than the pivot to its left
 - Move all elements larger than the pivot to its right



2. Look for 2 items:

- **itemFromLeft** that is larger than the pivot
- **itemFromRight** that is smaller than the pivot

It is a Swap!

Quick sort

- Another popular algorithm that is also order of $n(\log n)$
- Core idea:
 - Pick an element: we'll call it the pivot
 - Move all elements smaller than the pivot to its left
 - Move all elements larger than the pivot to its right

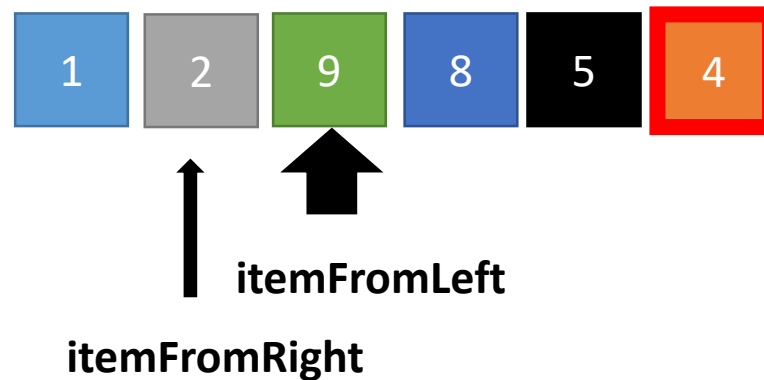


2. Look for 2 items:

- **itemFromLeft** that is larger than the pivot
- **itemFromRight** that is smaller than the pivot

Quick sort

- Another popular algorithm that is also order of $n(\log n)$
- Core idea:
 - Pick an element: we'll call it the pivot
 - Move all elements smaller than the pivot to its left
 - Move all elements larger than the pivot to its right



2. Look for 2 items:

- **itemFromLeft** that is larger than the pivot
- **itemFromRight** that is smaller than the pivot

4. When index of **itemFromLeft** > index of **itemFromRight**, then we swap pivot with **itemFromLeft**.

Quick sort

- Another popular algorithm that is also order of $n(\log n)$
- Core idea:
 - Pick an element: we'll call it the pivot
 - Move all elements smaller than the pivot to its left
 - Move all elements larger than the pivot to its right



Now the **pivot** is in the correct position!

Quick sort

- Another popular algorithm that is also order of $n(\log n)$
- Core idea:
 - Pick an element: we'll call it the pivot
 - Move all elements smaller than the pivot to its left
 - Move all elements larger than the pivot to its right



QuickSort is a recursive function – let's apply it on the right part of the list.

Quick sort

- Another popular algorithm that is also order of $n(\log n)$
- Core idea:
 - Pick an element: we'll call it the pivot
 - Move all elements smaller than the pivot to its left
 - Move all elements larger than the pivot to its right



QuickSort is a recursive function – let's apply it on the right part of the list.

Quick sort

- Another popular algorithm that is also order of $n(\log n)$
- Core idea:
 - Pick an element: we'll call it the pivot
 - Move all elements smaller than the pivot to its left
 - Move all elements larger than the pivot to its right



QuickSort is a recursive function – let's apply it on the right part of the list.

Quick sort

- Another popular algorithm that is also order of $n(\log n)$
- Core idea:
 - Pick an element: we'll call it the pivot
 - Move all elements smaller than the pivot to its left
 - Move all elements larger than the pivot to its right



QuickSort is a recursive function – let's apply it on the right part of the list.

Quick sort

- Another popular algorithm that is also order of $n(\log n)$
- Core idea:
 - Pick an element: we'll call it the pivot
 - Move all elements smaller than the pivot to its left
 - Move all elements larger than the pivot to its right



QuickSort is a recursive function – let's apply it on the right part of the list.

Quick sort

- But how do you select the pivot?
 - First item on the list
 - Last item on the list
 - The middle item
 - The median of the first, last, and middle item.
 - Or randomly!
- Time complexity
 - Best case: $O(n \log n)$
 - Worst case: $O(n^2)$ → e.g., if you pivot is the minimum element every time

Files in Python

From Sli.do: can you go over file handling and what you're allowed to do to a file in python for the revision lecture please

Writing the first file

- Opening a file creates a **file object** (in this case, “myfile”), and makes a file available for reading or writing. For a file that is open, accessing its **file object** accesses the file:
- “r” → opens file in **read** mode (default).
- “w” → opens file in **write** mode, overwrites existing content.
- “a” → opens file in **append** mode, appends to the end of the file.
- “x” → creates the specified file, returns an error if the file exists.
- “t” → opens file as **text** (default)
- “b” → opens file as **binary**

```
myfile = open("data.txt", "w")
myfile.write("Writing my first file, yay! (I guess)\n")
myfile.write("-----\n")
myfile.write("Hello, earthlings!\n")
myfile.close()
```

File *Streams*

- Unlike compound datatypes (e.g., lists, dictionaries, etc.), files are **not** a collection of variables that you can read/write at any location.
- Files provide a **stream** of data.
- You can only work with file streams **in order**.
- This applies to both reading and writing.

Using Files

```
file1 = open(<filename>, <mode>) #open file in required mode
                                     #and return handle
file1.read()           #returns the entire contents, as a string

file1.read(10)         #returns the next 10 characters

file1.readline()       #returns the next line, as a string

file1.readlines()      #returns all of the remaining lines,
                        #as a list of strings

file1.close()          #When we have finished using the file
```

Using Files

- A good practice is to use the **with** keyword when dealing with files
 - It closes the file even if you don't explicitly close it
 - It closes it even if an *exception* is thrown
 - It is important to close files not being used
 - **Context Managers** help us achieve it; we use the **with** keyword to use them
 - `open()` is a built-in function that returns a context manager

```
#without with
>>> f1 = open('data.txt', 'r')
>>> read_data = f1.read()

>>> f1.closed
False
```

```
#with with
>>> with open('data.txt', 'r') as f2:
...     read_data = f2.read()

#we have gone out of scope of the "with"
block here, so the resource (f2) is
automatically "cleaned up"

>>> f2.closed
True
```

Using Files

- If we have a file that is open for writing, we can invoke the `write()` function on the file handle to write a string.
- Or `writelines()`, while supplying a list of strings

```
#open and write two lines
with open("test.txt", "w") as outFile:
    outFile.write("this is line 1")
    outFile.write("this is line 2")

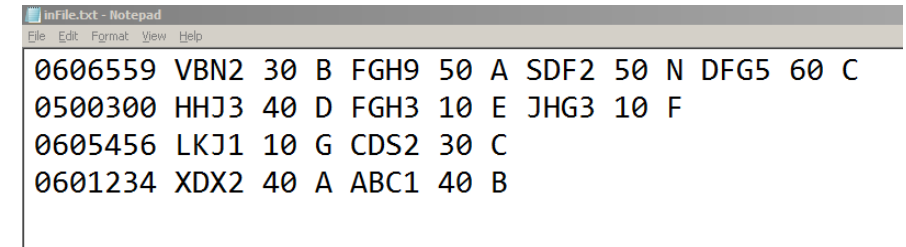
#read
with open("test.txt", "r") as inFile:
    print(inFile.readline())

this is line 1this is line 2
```

Some Useful Tasks

- **Processing contents of files, line by line**

```
with open('inFile.txt', 'r') as f:
    for line in f:
        print(line)
```



```
inFile.txt - Notepad
File Edit Format View Help
0606559 VBN2 30 B FGH9 50 A SDF2 50 N DFG5 60 C
0500300 HHJ3 40 D FGH3 10 E JHG3 10 F
0605456 LKJ1 10 G CDS2 30 C
0601234 XDX2 40 A ABC1 40 B
```

- In the text file, all lines end with the newline character `\n`, and when we use `print`, it inserts a newline after every print command.
- This means the code above will print empty lines in between each line in the file

```
with open('inFile.txt', 'r') as f:
    for line in f:
        print(line.strip())
```

- the `strip()` command strips whitespace characters (which includes `\n`) from beginning and end of string.
- This prints each line in the file without introducing an additional line break.

Some Useful Tasks

- Handling files **simultaneously** with **with**

```
#open 3 files concurrently
#one for reading, two for writing

with open('inFile.txt', 'r') as a, \
    open('outFile1.txt', 'w') as b, \
    open('outFile2.txt', 'w') as c:

    #enumerate* iterates over all the lines
    #and gives us their index as well
    for i, line in enumerate(a):

        #odd lines go to b
        if i%2:
            b.write(line)

        #even to a
        else:
            c.write(line)
```

Python Escape Sequences (partial)

- Used to signal an **alternative interpretation** of a series of characters
- Solve the problem of using **special characters** inside a string declaration

For this	Use this	Setting string to:	Printing string yields:
'	\'	'Don\'t do that'	Don't do that
"	\"	"She said \"hi\""	She said "hi"
\	\\	"Backslash: \\"	Backslash: \
[newline]	\n	"1\n2"	1 2
[horizontal tab]	\t	"1\t2"	1 2
...

Exception handling

Exceptions: Understanding the “Traceback”


```
#%% Exceptions, show traceback in action

def funcA():
    print("I am in funcA(), and I'm about to do something wicked")
    print(100/0)

def funcB():
    print("I am in funcB(), about to call funcA()")
    funcA()

def funcC():
    print("I am in funcC(), about to call funcB()")
    funcB()

#call funcC --> funcB --> funcA --> exception!
funcC()
```



```
I am in funcC(), about to call funcB()
I am in funcB(), about to call funcA()
I am in funcA(), and I'm about to do something wicked
Traceback (most recent call last):

  File "<ipython-input-7-be079daae2ac>", line 17, in <module>
    funcC()

  File "<ipython-input-7-be079daae2ac>", line 13, in funcC
    funcB()

  File "<ipython-input-7-be079daae2ac>", line 8, in funcB
    funcA()

  File "<ipython-input-7-be079daae2ac>", line 3, in funcA
    print(100/0)

ZeroDivisionError: division by zero
```


Errors and Exceptions

- There are three main categories of errors:
 - Syntax errors
 - the interpreter cannot parse the statements because they are not well formed.

```
>>> print("Hello World"  
SyntaxError  
>>> print ("Hello World"  
...  
...  
... )  
Hello World
```

Errors and Exceptions

- There are three main categories of errors:
 - Syntax errors
 - the interpreter cannot parse the statements because they are not well formed.
 - Run-time errors
 - The interpreter can parse the program, but an illegal operation happens during run-time

```
>>> 3/0
```

```
ZeroDivisionError: integer division or modulo by zero
```

```
>>> [1,2,3][3]
```

```
IndexError: list index out of range
```

Errors and Exceptions

- There are three main categories of errors:
 - Syntax errors
 - the interpreter cannot parse the statements because they are not well formed.
 - Run-time errors → aka Exceptions – these ones can be handled
 - The interpreter can parse the program, but an illegal operation happens during run-time
 - Semantic errors
 - The interpreter can parse the program and it is able to run it. But it does not produce what the programmer has intended.

```
>>> print ("Adding 6 to 7 results in", 6+6)
12 # The programmer was expecting 13
```

Handling Run-time errors (aka Exceptions)

- Why?
 - Run-time errors are not always clear from the code.
 - They might happen due to the user's input, a file's content, or variable manipulations that had unexpected outcomes.
- What happens when a run-time error occurs?
 - Program stops
 - Python prints out the **traceback**

```
In [1]: print(150/0)
Traceback (most recent call last):

  File "<ipython-input-1-5df4ebc3d506>", line 1, in <module>
    print(150/0)

ZeroDivisionError: division by zero
```

```
In [3]: a = []
...: print(a[1])
...:
Traceback (most recent call last):

  File "<ipython-input-3-b8f79104f453>", line 2, in <module>
    print(a[1])

IndexError: list index out of range
```

Handling Run-time errors (aka Exceptions)

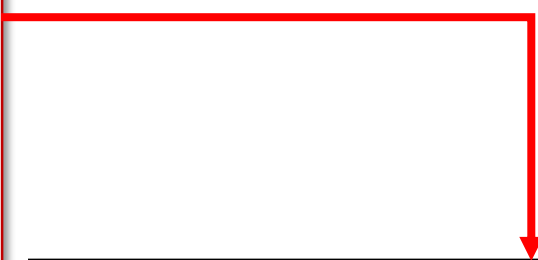
```
### Exceptions, show traceback in action

def funcA():
    print("I am in funcA(), and I'm about to do something wicked")
    print(100/0)

def funcB():
    print("I am in funcB(), about to call funcA()")
    funcA()

def funcC():
    print("I am in funcC(), about to call funcB()")
    funcB()

#call funcC --> funcB --> funcA --> exception!
funcC()
```



```
I am in funcC(), about to call funcB()
I am in funcB(), about to call funcA()
I am in funcA(), and I'm about to do something wicked
Traceback (most recent call last):

  File "<ipython-input-7-be079daae2ac>", line 17, in <module>
    funcC()

  File "<ipython-input-7-be079daae2ac>", line 13, in funcC
    funcB()

  File "<ipython-input-7-be079daae2ac>", line 8, in funcB
    funcA()

  File "<ipython-input-7-be079daae2ac>", line 3, in funcA
    print(100/0)

ZeroDivisionError: division by zero
```

Handling Run-time errors (aka Exceptions)


```
## Exceptions, nested functions, more graceful exit (Try / Except)

def funcA():
    print("I am in funcA(), and I'm about to do something wicked")
    try:
        print(100/0)
    except:
        print("Well, at least I tried!")

def funcB():
    print("I am in funcB(), about to call funcA()")
    funcA()

def funcC():
    print("I am in funcC(), about to call funcB()")
    funcB()

#call funcC --> funcB --> funcA --> exception!
funcC()
```



```
I am in funcC(), about to call funcB()
I am in funcB(), about to call funcA()
I am in funcA(), and I'm about to do something wicked
Well, at least I tried!
```

- We add a **try/except** block to funcA:
 - **Try** executes statements in the **try** block
 - If **no exception** occurs, **except** block is ignored
 - If an **exception** occurs, execution jumps to **except**, and then continues

Handling Run-time errors (aka Exceptions)

- What if we don't want the program to stop but to take a corrective action instead?
- We can handle the exception by adding `try/except` blocks

```
def access_file():  
    filename = input('Enter a filename: `')  
    with open(filename, "r") as file1:  
        # do something with the contents of the file  
        print('file %s opened (and accessed) OK' %filename)
```

- A runtime error could occur if the user inputs a file name that does not exist.

Handling Run-time errors (aka Exceptions)

- What if we don't want the program to stop but to take a corrective action instead?
- We can handle the exception by adding **try/except** blocks

```
def access_file():  
    filename = input('Enter a filename: ')  
    try:  
        with open(filename, "r") as file1:  
            # do something with the contents of the file  
            print('file %s opened (and accessed) OK' %filename)  
    except:  
        print('There is no such file: ', filename)  
        print('doing something else instead... ')
```

- **Try** executes statements in the first block
 - If no exception occurs, except block is ignored
 - If an exception occurs, execution jumps straight in **except**, and then continues

Handling Run-time errors (aka Exceptions)

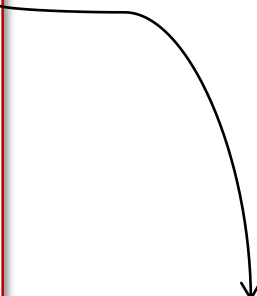
- Let's say we want to **insist** that the user enter a correct filename.
- That is, we want the user to **keep entering** filenames until he/she enters a correct one.

```
#!/usr/bin/env python3
# Insist the user enters a correct filename

def access_file(filename):
    accessed = True
    try:
        with open(filename, "r") as file1:
            # do something with the contents of the file
            # "pass" is a null (do nothing) operation
            pass
    except:
        accessed = False
    return accessed

def insist():
    # initialize as False...
    accessed = False

    # ... and keep trying until you get a True from access_file()
    while not accessed:
        filename = input('Enter a filename: ')
        accessed = access_file(filename)
    print('file %s exists and accessed OK' % filename)
```



```
In [6]: insist()
Enter a filename: hello
Enter a filename: hello.dat
Enter a filename: hello.txt
file hello.txt exists and accessed OK
```

Handling Run-time errors (aka Exceptions)

- You can handle different types of errors

```
import sys
```

```
try:
```

```
    f = open('myfile.txt')
```

```
    s = f.readline()
```

```
    I = int(s.strip())
```

```
except IOError as e:
```

```
    print "I/O error({0}): {1}".format(e.errno, e.strerror)
```

```
except ValueError:
```

```
    print ("There are invalid integers in this file.")
```

```
except:
```

```
    print ("Unexpected error")
```

Handling Run-time errors (aka Exceptions)

- You can handle multiple error types using the same except clause

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    I = int(s.strip())
except (IOError, ValueError) :
    print "An I/O or ValueError has occurred")
```

Try, Except, and Finally

- Want to run some code **regardless** of whether an exception is previously raised or not
- We want to **try** something:
 - then do something if an exception is raised → **except**
 - and/or do something **regardless of** whether or not an exception is raised → **finally**
- Useful when e.g. **managing resources** like a file, which we want to close whether or not an exception is raised

```
#try to execute funcA()  
try:  
    funcA()  
  
#if funcA() raised an exception, call funcB  
except:  
    funcB()  
  
#call funcC() whether or not funcA raised an exception  
finally:  
    funcC()
```

- A try block should have **at least** either an **except** or a **finally** block

Try, Except, and Finally

- When does it make a difference compared to simply running `funcC()` outside the `try/except` clause?
 - It makes a difference if you use `return` in the `except` clause

```
try:
    funcA()
except:
    funcB()
    return None # The finally block is run before the method returns
finally:
    funcC()
```

```
try:
    funcA()
except:
    funcB()
    return None
funcC() # This doesn't run if there's an exception.
```

Try, Except, and Finally

- When does it make a difference compared to simply running `funcC()` outside the `try/except` clause?
 - It makes a difference if you use `return` in the `except` clause
 - If an exception is thrown in `funcA()` that is not a `TypeError`, the code in `finally` will run, but the code afterwards won't run.

```
try:
    funcA()
except TypeError: # This will run only if there is a TypeError
    funcB()
finally:
    other_code() # This will always run

other_code2() # This won't run if funcA() has an error that is not TypeError
```

Try, Except, and Finally

- When does it make a difference compared to simply running `funcC()` outside the `try/except` clause?
 - It makes a difference if you use `return` in the `except` clause
 - If an exception is thrown in `funcA()` that is not a `TypeError`, the code in `finally` will run, but the code afterwards won't run.
 - If an exception is thrown in the `except` part, the code in `finally` will always run.

Try, Except, and Finally + else

- `finally` is executed regardless of whether the statements in the `try` block fail or succeed. `else` is executed only if the statements in the `try` block don't raise an exception.

```
while True:
    try:
        age=int(eval(input('Enter your age: ')))
    except (NameError,ValueError):
        print("Invalid value")
    else:
        if age <= 21:
            print("Your ticket costs £11.")
        else:
            print("Your ticket costs £15.")
    finally:
        print("Resetting program..")
    print("All errors were handled")
```


Try, Except, and Finally + else

- `finally` is executed regardless of whether the statements in the `try` block fail or succeed. `else` is executed only if the statements in the `try` block don't raise an exception.

```
while True:
    try:
        age=int(eval(input('Enter your age: ')))
    except (NameError,ValueError):
        print("Invalid value")
    else:
        if age <= 21:
            print("Your ticket costs £11.")
        else:
            print("Your ticket costs £15.")
    finally:
        print("Resetting program..")
        print("All errors were handled")
```

```
Enter your age: a
Invalid value
Resetting program..
All errors were handled
```

```
Enter your age: 13
Your ticket costs £11
Resetting program..
All errors were handled
```

```
Enter your age: 10/0
Resetting program..
ZeroDivisionError: integer division or modulo
by zero
```

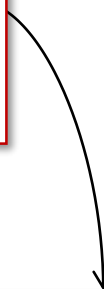
Raising your own exceptions

- Can a program **deliberately** create its **own** exceptions?
- If our program detects what *it* considers an error, we **can** raise an exception.
- Python provides us some **built-in exception types*** which we can match to the kind of error we want to raise.
 - **TypeError**: Operation or function is applied to an object of inappropriate type.
 - **ValueError**: Right type, but inappropriate value.

* For a description of all built-in exception types
<https://docs.python.org/3/library/exceptions.html>

Raising your own exceptions: example

```
def get_income():  
    income = int(input("Please enter your pre-tax income: "))  
    if income < 0:  
        #Create a Value Error, and raise it (can be done in one step too)  
        error = ValueError("{0} is not valid income".format(income))  
        raise error  
    return income
```



```
In [14]: get_income()  
Please enter your pre-tax income: -100  
Traceback (most recent call last):  
  
  File "<ipython-input-14-d754c339808a>", line 1, in <module>  
    get_income()  
  
  File "<ipython-input-13-1a727b4ae92e>", line 6, in get_income  
    raise error  
ValueError: -100 is not valid income
```

Raising your own exceptions: example 2

```
#find the maximum value from a list
def maxList(x):
    max = x[0]
    for v in x:
        if v > max:
            max = v
    return max
```

- If we call it with an empty list →

```
In [17]: maxList([])
Traceback (most recent call last):

File "<ipython-input-17-680e209cded0>", line 1, in <module>
    maxList([])

File "<ipython-input-15-da6ecca4a378>", line 2, in maxList
    max = x[0]

IndexError: list index out of range
```

- If we call it with a non-list → argument (e.g., an **int**)

```
In [16]: maxList(1)
Traceback (most recent call last):

File "<ipython-input-16-0fb8890f2fb8>", line 1, in <module>
    maxList(1)

File "<ipython-input-15-da6ecca4a378>", line 2, in maxList
    max = x[0]

TypeError: 'int' object is not subscriptable
```

Raising your own exceptions: example 2

```
#find the maximum value from a list  
def maxList(x):  
    max = x[0]  
    for v in x:  
        if v > max:  
            max = v  
    return max
```

- What if we call it with a string argument? →

```
In [19]: maxList('blah')  
Out[19]: 'l'
```

- It is not throwing an exception, BUT
- It is not behaving the way we had intended
 - (assuming our intention was to not use this function for strings)

Raising your own exceptions: example 2

- When we give our function an input **that is not intended for it** (i.e., intended by us, the programmer):
 - Sometimes the Python's built-in exception handler will throw an exception that is not very informative/helpful.
 - Sometimes, it doesn't (but is not doing what it was supposed to be doing).
 - How can we ensure consistent responses?

Raising your own exceptions: example 2

- We can do the checks and **raise** more informative exceptions

```
def maxList(x):  
    if type(x) is not list:  
        raise TypeError('%s is not a list' % x)  
    if len(x) == 0:  
        raise ValueError('%s is an empty list' % x)  
  
    max = x[0]  
    for v in x:  
        if v > max:  
            max = v  
    return max
```

Raising your own exceptions: example 2

```
In [17]: maxList([])
Traceback (most recent call last):

  File "<ipython-input-17-680e209cded0>", line 1, in <module>
    maxList([])

  File "<ipython-input-15-da6ecca4a378>", line 2, in maxList
    max = x[0]

IndexError: list index out of range
```

```
In [16]: maxList(1)
Traceback (most recent call last):

  File "<ipython-input-16-0fb8890f2fb8>", line 1, in <module>
    maxList(1)

  File "<ipython-input-15-da6ecca4a378>", line 2, in maxList
    max = x[0]

TypeError: 'int' object is not subscriptable
```

```
In [19]: maxList('blah')
Out[19]: 'l'
```

```
In [48]: maxList([])
Traceback (most recent call last):

  File "<ipython-input-48-aef2a216c918>", line 1, in <module>
    maxList([])

  File "<ipython-input-46-ad5178a89cd1>", line 5, in maxList
    raise ValueError('%s is an empty list' % x)

ValueError: [] is an empty list
```

```
In [47]: maxList(1)
Traceback (most recent call last):

  File "<ipython-input-47-0fb8890f2fb8>", line 1, in <module>
    maxList(1)

  File "<ipython-input-46-ad5178a89cd1>", line 3, in maxList
    raise TypeError('%s is not a list' % x)

TypeError: 1 is not a list
```

```
In [49]: maxList('blah')
Traceback (most recent call last):

  File "<ipython-input-49-b5f4ad5e59ce>", line 1, in <module>
    maxList('blah')

  File "<ipython-input-46-ad5178a89cd1>", line 3, in maxList
    raise TypeError('%s is not a list' % x)

TypeError: blah is not a list
```


Wait.. Why not use if conditions instead of exception handling?

- You can approach errors in two ways:
 - Exception handling – it is **Easier to Ask for Forgiveness than Permission (EAFP)**
 - assumes the existence of valid keys or attributes and catches exceptions if the assumption proves false.
 - clean and fast style; is characterized by the presence of many `try` and `except` statements.
 - Defensive programming – **Look Before You Leap (LBYL)**
 - explicitly `test` for `pre-conditions` before making calls or lookups.
 - characterized by the presence of many `if` statements.
 - In multi-threaded environments (parallel **threads** of execution), LBYL risks suffering from `race conditions` between “the looking” and “the leaping”.
 - A **looks** at shared resource.
 - B changes that resource.
 - A **leaps** (i.e., uses that resource).
 - A (potentially) crashes because B changed the resource in-between the **look** and **leap** stages.

Python encourages EAFP coding style – see
<https://docs.python.org/3.6/glossary.html#term-eafp>

Exception handling vs defensive programming

- Exception handling is **cleaner**
 - Can **delegate** responsibility for handling the error (e.g., raise exception).
 - Do **not have to handle it locally** (much more modular).
- Performance **penalty** when an exception is raised.
 - Considerable **information needs to be gathered** in the exception.
 - Gathering that information takes time.
 - So, if:
 - error very often: **defensive programming** (avoid exceptions)
 - error case occurs rarely (i.e., it is an “exception”): then **don’t pay the penalty** of always checking with if/else – don’t be defensive – instead, use exception handling.

Other topics

Higher order Functions

- Higher order functions take functions as parameters and apply them on iterables.
- We will cover three of them:
 - `map()`
 - `filter()`
 - `reduce()`

Higher order functions: map()

- Takes another **function** as a parameter, and an **iterable** (e.g., list, dictionary, set or tuple).
- Applies the **function** on each element of the **iterable**

```
def newfunc(a):  
    return a*a
```

```
x = map(newfunc, [1,2,3,4]) #x is the map object  
print(list(x)) # outputs [1,4,9,16]
```

- You can use the `lambda` command to shorten this even more

```
>>> x = map(lambda a: a*a, [1,2,3,4])  
[1, 4, 9, 16]
```

Higher order functions: filter()

- Takes another **function** as a parameter, and an **iterable** (e.g., list, dictionary, set or tuple).
- Checks each element of the iterable against a condition, and returns those the evaluate to `True`

```
def newfunc(a):  
    return a % 2 == 0  
  
x = filter(newfunc, [1,2,3,4]) #x is the filter object  
print(list(x)) # outputs [2,4]
```

- You can use the `lambda` command to shorten this even more

```
>>> x = filter(lambda a: a % 2 == 0, [1,2,3,4])  
[2,4]
```

Higher order functions: reduce()

- Takes another **function that has two parameters** as a parameter, and an **iterable** (e.g., list, dictionary, set or tuple).
- Applies an expression to the iterables consecutively.
- Unlike `map()` and `filter()`, it returns a value rather than an object.
- Needs to be imported from the `functools` module

```
from functools import reduce
def newfunc(a,b):
    return a + b
```

```
x = reduce(newfunc, [1,2,3,4]) #x is the map object
print(x) # outputs 10, which is 1+2+3+4
```

- You can use the `lambda` command to shorten this even more

```
>>> x = reduce(lambda a,b: a + b, [1,2,3,4])
10
```

Lambda vs list comprehension

- list comprehension achieves the same goal as lambda

```
>>> [x for x in [1,2,3,4,10] if x%2 == 0]  
[2, 4, 10]
```

```
>>> list(filter(lambda x: x%2 == 0, [1,2,3,4,10]))  
[2, 4, 10]
```

- list comprehension is
 - slightly faster than lambda + filter() or lambda + map()
 - saves you from having to cast the filter object to a list
 - It is also neater

List comprehension

```
>>> [x+1 for x in [1,2,3]]  
[2,3,4]
```

```
>>> [x[0] for x in [["Feb",12],["Mar",2]]]  
["Feb","Mar"]
```

```
>>> [float(x[1]) for x in  
[["Feb",12],["Mar",2]]]  
[12.0,2.0]
```