

# Computing Foundations II

Computer Science for  
Lawyers and Humanitarian Workers

Session 2

Prof. Bryan Ford  
Decentralized/Distributed Systems (DEDIS)  
EPFL

Moodle:

<https://moodle.epfl.ch/course/view.php?id=15667>

# Session 2 Outline

- Part 1: Basics of Programming
  - via a brief “crash course” in the **Python** language
- Part 2: Basics of Algorithms and Complexity
  - searching, sorting, organizing data, big-O notation

# Part 1: Basics of Programming

Look inside ↓

## Python FOR BEGINNERS

A Crash Course Guide To  
Learn Python in ~~1 Week~~

1 Hour :)

# Introducing Python

A popular *high-level language* that hides many details in order to be simple to learn and use

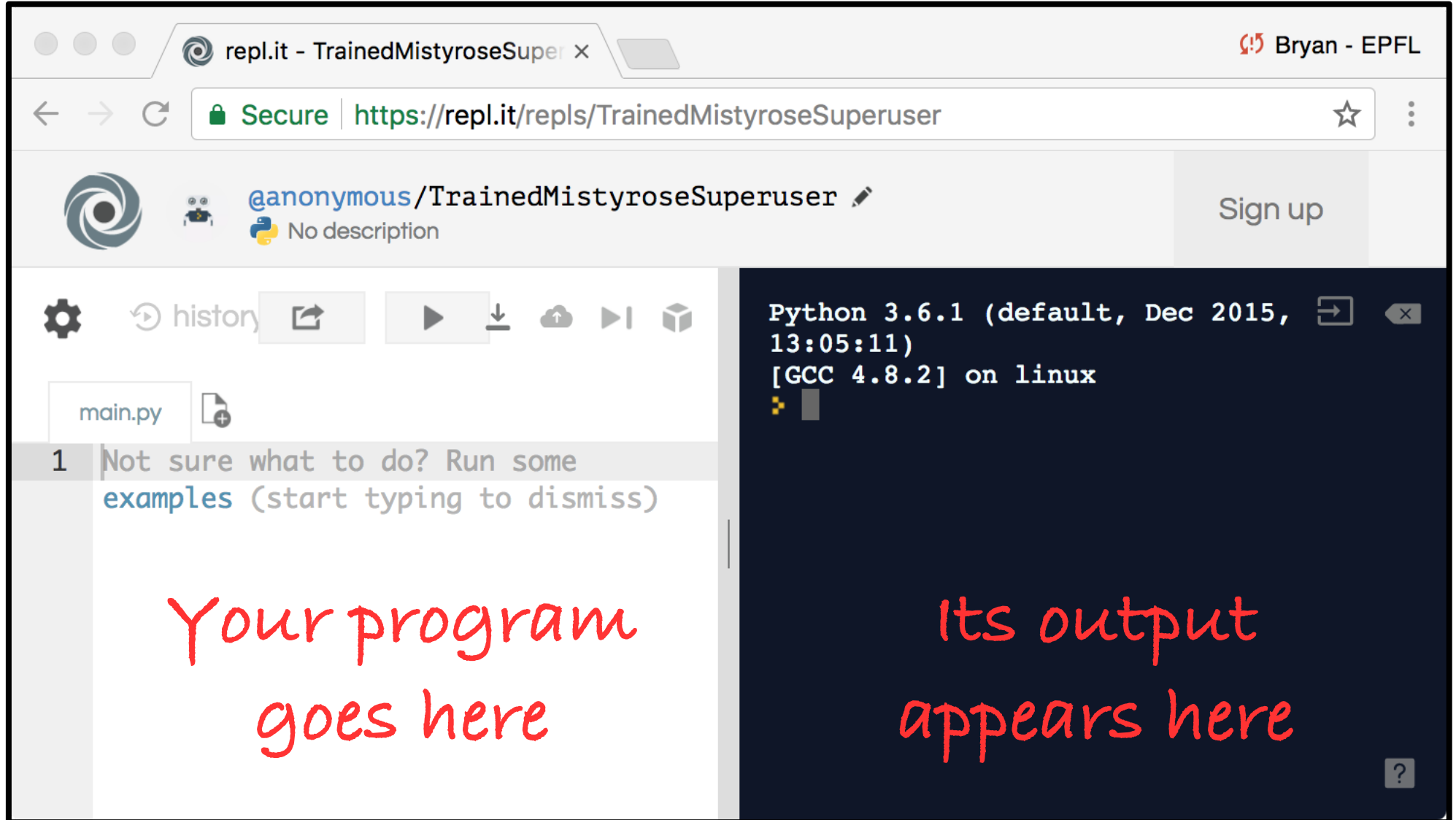
- But powerful due to rich ecosystem of add-ons
- Especially popular for statistics & data science

But principles we cover are *not specific to Python*

We will use a web-based Python interpreter:

- Go to <https://repl.it/> and search for 'Python3'
- Or just click: <https://repl.it/languages/python3>

# You should see something like...



The screenshot shows a web browser window with the URL <https://repl.it/repls/TrainedMistyroSuperuser>. The page header includes the Repl.it logo, the username `@anonymous/TrainedMistyroSuperuser`, and a "Sign up" button. Below the header, there is a toolbar with icons for settings, history, share, run, download, upload, and a 3D cube. The code editor on the left shows a file named `main.py` with the following text:

```
1 Not sure what to do? Run some  
  examples (start typing to dismiss)
```

Handwritten red text "Your program goes here" is overlaid on the code editor. The terminal on the right shows the following output:

```
Python 3.6.1 (default, Dec 2015,  
13:05:11)  
[GCC 4.8.2] on linux  
✖
```

Handwritten red text "Its output appears here" is overlaid on the terminal. A small question mark icon is visible in the bottom right corner of the terminal area.

# Hello World in Python

In computer science tradition, your first program in *any* language should print “Hello World”

Enter this program in the ‘main.py’ window at left:

```
print("Hello World")
```

...then click the  
“Play button”:



# Data Types: Integers, Strings, ...

Python and other high-level languages help by distinguishing between different *data types*

- Example: “Hello World” (with quotes) is a **string**
- Example: 1234 (with no quotes) is an **integer**

You can **print** most any data type:

```
print(3, "blind mice")
```

But they have important differences as we'll see!

# Arithmetic Expressions

Python (like most high-level languages) lets you use *expressions* to convey complex calculations.

Example:

```
print(2*3+4)
```

Python breaks this code into simpler operations:

- 1) Multiply the integer 2 by the integer 3
- 2) Add the result of step 1 to the integer 4
- 3) Print the result of step 2



# Arithmetic Operators

You do integer arithmetic by combining integers using a variety of standard operators...

## Arithmetic Operators

...but some use funny symbols because math operators like divide ( $\div$ ) aren't on most keyboards

Operator	Meaning	Example
<b>+</b>	Addition	$4 + 7 \longrightarrow 11$
<b>-</b>	Subtraction	$12 - 5 \longrightarrow 7$
<b>*</b>	Multiplication	$6 * 6 \longrightarrow 36$
<b>/</b>	Division	$30 / 5 \longrightarrow 6$
<b>%</b>	Modulus	$10 \% 4 \longrightarrow 2$
<b>//</b>	Quotient	$18 // 5 \longrightarrow 3$
<b>**</b>	Exponent	$3 ** 5 \longrightarrow 243$

# Strings

Anything in single- or double-quotes is a *string*

- Example: “Hello World” in our first program

The quotes are critical: they tell Python *not* to treat their contents like code and try to compute them!

Example:

```
print("2*3+4", "is", 2*3+4)
```

We’re printing 3 things: 2 strings and an integer

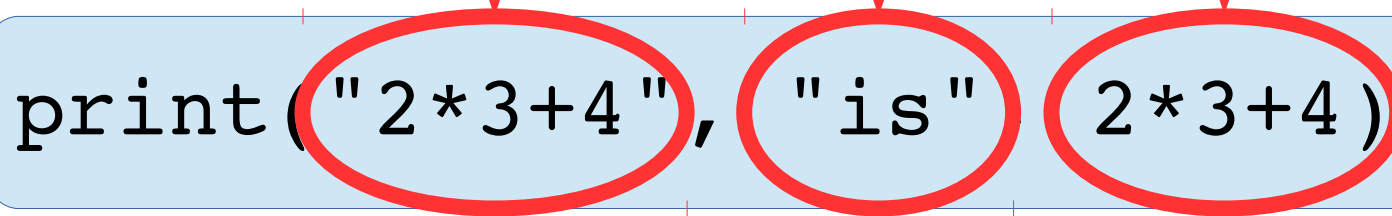
# Strings

Anything in single- or double-quotes is a *string*

- Example: “Hello World” in our first program

The quotes are critical: they tell Python *not* to treat their contents like code and try to compute them!

Example:



The diagram shows a Python print statement: `print("2*3+4", "is", 2*3+4)`. The code is enclosed in a light blue rounded rectangle. Above the rectangle, the word *strings* is written in red, with two red arrows pointing down to the first two arguments, `"2*3+4"` and `"is"`. The word *integer* is written in red above the third argument, `2*3+4`, with a red arrow pointing down to it. Each of the three arguments is also circled in red.

```
print("2*3+4", "is", 2*3+4)
```

We're printing 3 things: 2 strings and an integer

# String Operators

Arithmetic isn't just for numbers!

- You can *concatenate* strings with '+' operator

Notice the difference in this example:

```
print(1+2, 'is integer arithmetic')  
print('1'+'2', 'is string arithmetic')
```

Python uses *different operators on different types*

# Bit Isn't Data “Just Bits and Bytes”?

Yes. Computer memory is just an array of bytes:



But languages like Python organize memory using *metadata* to distinguish high-level data types...



tag: means  
“this is an  
integer”

number of  
digits in the  
integer

the actual  
integer

tag  
“this is a  
string”

length of  
the string

ASCII  
encoded  
characters

# All is *encoded* into bits and bytes

Example: characters encoded via the ASCII table

## ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	\$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

...or international-friendly Unicode and UTF8

# Abstracting Data with Variables

We need a way to make program code *abstract*, so we can rerun the *same code* on *different data*.

**Variables** are the most basic abstraction tool

- Act as “holes” for data to be filled in elsewhere

For example, try:

```
num = 5
print(num, 'plus', num, 'is', num+num)
```

‘num’ is a variable we can refer to several times

- Change in one place to process different data

# Abstracting Code with Functions

We need to reuse *program code* as well as data.

**Functions** let us name a fragment of code once, then reuse it multiple times by invoking that name:

```
def eeny():  
    print("Eeny, meeny, miny, moe")  
  
eeny()  
print("Catch a tiger by the toe")  
print("If he hollers let him go")  
eeny()
```



# Abstracting Code with Functions

We need to reuse *program code* as well as data.

**Functions** let us name a fragment of code once, then reuse it multiple times by invoking that name:

```
def eeny():  
    print("Eeny, meeny, miny, moe")  
  
eeny()  
print("Catch a tiger by the toe")  
print("If he hollers let him go")  
eeny()
```

function definition →

function name →

function calls →

function calls →

# Reusing Code with Different Data

Functions wouldn't be so useful if they always had to repeat *exactly* the same code on the same data

**Parameters** are special variables allowing us to give a function different input data at each call:

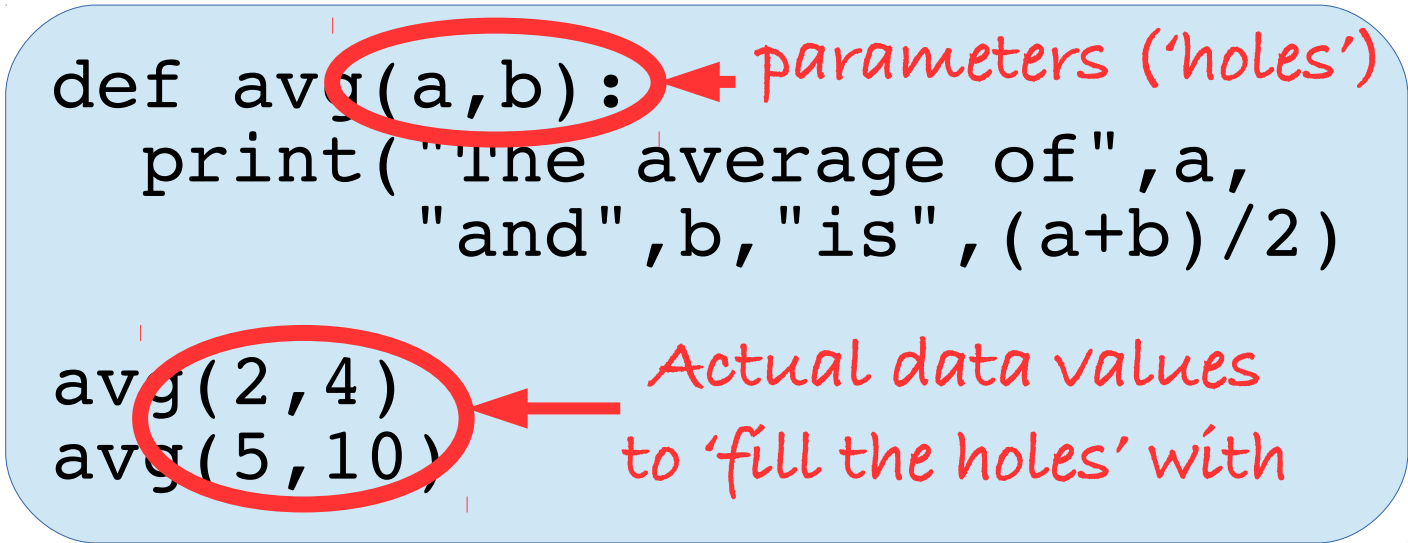
```
def avg(a,b):  
    print("The average of",a,  
          "and",b,"is",(a+b)/2)  
  
avg(2,4)  
avg(5,10)
```

We can now reuse 'avg' on any pair of numbers

# Reusing Code with Different Data

Functions wouldn't be so useful if they always had to repeat *exactly* the same code on the same data

**Parameters** are special variables allowing us to give a function different input data at each call:



```
def avg(a,b):  
    print("The average of",a,  
          "and",b,"is", (a+b)/2)  
  
avg(2,4)  
avg(5,10)
```

The diagram illustrates the concept of function parameters. It shows a function definition `def avg(a,b):` and two function calls `avg(2,4)` and `avg(5,10)`. Red circles highlight the parameter list `(a,b)` in the definition and the argument lists `(2,4)` and `(5,10)` in the calls. Red arrows point from the text "parameters ('holes')" to the first circle and from "Actual data values to 'fill the holes' with" to the second circle.

parameters ('holes')

Actual data values to 'fill the holes' with

We can now reuse 'avg' on any pair of numbers

# Returning Data from Functions

Functions can not only take but also produce data

- By 'return'ing it for use in the calling code

Example: 'avg' is more generic and useful if it lets the caller decide how to use the computed result.

- Assign to variable, use in further calculation...

```
def avg(a,b):  
    return (a+b)/2
```

```
toms_age = 20; bobs_age = 40  
avg_age = avg(toms_age,bobs_age)  
print("avg_age:", avg_age)
```

# Flow Control: Conditionals

Sometimes we need to run *different code paths* depending on the data a function is given.

Example: to compute the *maximum* of two numbers, we need a way to compare them!

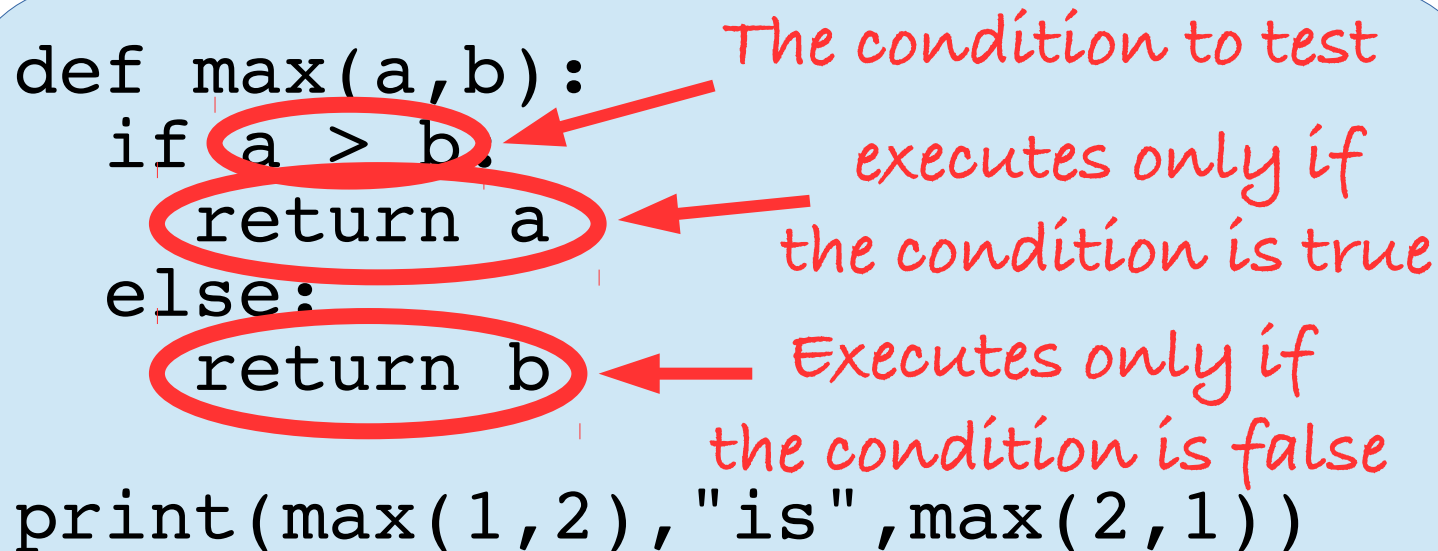
```
def max(a,b):  
    if a > b:  
        return a  
    else:  
        return b
```

```
print(max(1,2), "is", max(2,1))
```

# Flow Control: Conditionals

Sometimes we need to run *different code paths* depending on the data a function is given.

Example: to compute the *maximum* of two numbers, we need a way to compare them!



```
def max(a,b):  
    if a > b:  
        return a  
    else:  
        return b  
  
print(max(1,2), "is", max(2,1))
```

The condition to test  
executes only if  
the condition is true

Executes only if  
the condition is false

# Boolean Predicates

‘if’ can calculate complex conditional predicates by combining *relational* and *logical* operators

- **Relational:** compare things, yield true or false
- **Logical:** take true or false, yield true or false

## Relational Operators

Operator	Description
>	greater than
<	less than
==	equal to
<=	less or equal to
>=	greater or equal to
!=	lnot equal to

## Logical Operators

Operator	Description
<b>not</b> x	opposite of x
x <b>and</b> y	False unless x and y both True
x <b>or</b> y	True unless x and y both False

# Boolean Predicate Example

Example: who is and isn't legally of (unrestricted) working age in Switzerland?

```
def is_working_age(name, age):  
    if (age >= 18) and (age <= 65):  
        return name + " is working age"  
    else:  
        return name + " isn't working age"  
  
print(is_working_age("Alice", 15))  
print(is_working_age("Bob", 30))  
print(is_working_age("Charlie", 68))
```



# Boolean Predicate Example

Example: who is and isn't legally of (unrestricted) working age in Switzerland?

```
def is_working_age(name, age):  
    if (age >= 18) and (age <= 65):  
        return name + " is working age"  
    else:  
        return name + " isn't working age"  
  
print(is_working_age("Alice", 15))  
print(is_working_age("Bob", 30))  
print(is_working_age("Charlie", 68))
```

# Data Collections: Lists

We need programs to process *many* data items, without having to write code to handle each item!

**Collection** data types such as Python **lists** allow us to gather many small items into one composite

Example: a list of the members of a group

```
group = ["Alice", "Bob", "Charlie"]
print("the group contains", group)
print("there are", len(group), "members")
print("the first member is", group[0])
```

# Data Collections: Lists

We need programs to process *many* data items, without having to write code to handle each item!

**Collection** data types such as Python **lists** allow us to gather many small items into one composite

Example: a list of the members of a group

```
group = ["Alice", "Bob", "Charlie"]  
print("the group contains", group)  
print("there are", len(group), "members")  
print("the first member is", group[0])
```

Built-in function returning  
the length of a list

Returns a particular item  
from a list by position

# Flow Control: Loops

**Loops** allow us to process all the items in a list, in sequence, without caring how many there are

Example: a function listing all members of a group

```
def list_members(group):  
    for name in group:  
        print(name, "is a member")  
  
list_members(["Alice", "Bob", "Charlie"])
```

# Flow Control: Loops

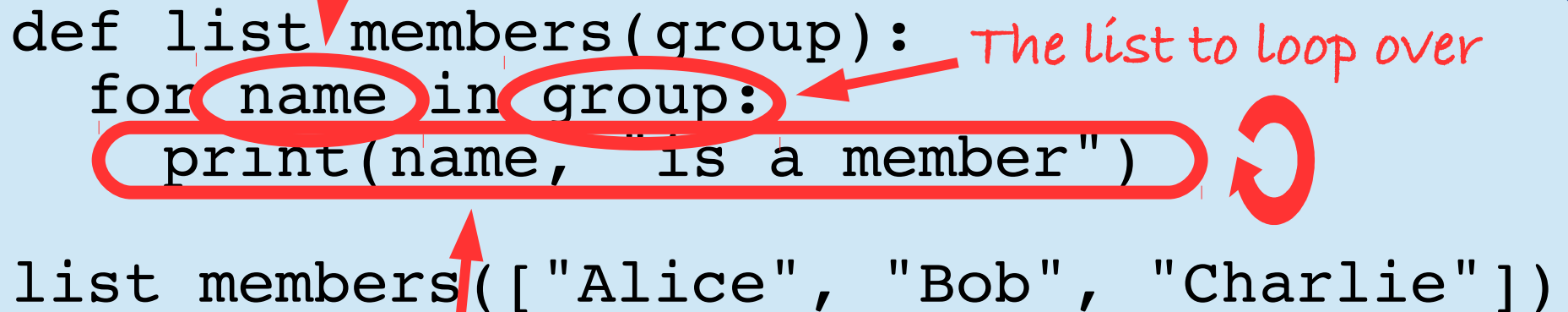
**Loops** allow us to process all the items in a list, in sequence, without caring how many there are

Example: a function listing all members of a group

*Temporary variable that refers to each element in turn*

```
def list_members(group):  
    for name in group:  
        print(name, "is a member")  
list_members(["Alice", "Bob", "Charlie"])
```

*The list to loop over*



*The code to run repeatedly, once for each list element*

# Lists of Anything and Everything

Lists can hold any data type, even other lists!

Example:

```
weird_list = [1, 2, "three", [4, "five"]]  
  
for item in weird_list:  
    print("next item:", item)  
print(len(weird_list), "items total")
```

Question: before running the above code, how many items total will the last line report?

Why?

# Calculating Aggregates over Lists

We can now calculate statistics over many integers without caring how many there are.

Example 1: find the average of a list of numbers

```
def avg_list(list):  
    sum = 0  
    for value in list:  
        sum = sum + value  
    return sum / len(list)  
  
print("avg", avg_list([10, 30, 20]))
```

# Calculating Aggregates over Lists

We can now calculate statistics over many integers without caring how many there are.

Example 1: find the average of a list of numbers

```
def avg_list(list):  
    sum = 0  
    for value in list:  
        sum = sum + value  
    return sum / len(list)
```

*Temporary variable*

*holding the running sum*

*Repeat for each value  
to add them up*

*Calculate the average*

```
print("avg", avg_list([10, 30, 20]))
```



# Calculating Aggregates over Lists

We can now calculate statistics over many integers without caring how many there are.

Example 2: find the maximum number in a list

```
def max_list(list):  
    largest = 0  
    for value in list:  
        if value > largest:  
            largest = value  
    return largest  
  
print("max", max_list([10, 30, 20]))
```

Can you change this into a 'min\_list' quickly?

# Calculating Aggregates over Lists

We can now calculate statistics over many integers without caring how many there are.

Example 2: find the maximum number in a list

```
def max_list(list):
```

```
    largest = 0
```

```
    for value in list:
```

```
        if value > largest:
```

```
            largest = value
```

```
    return largest
```

```
print("max", max_list([10, 30, 20]))
```

*Temporary variable  
holding the largest value  
we've seen so far*

*We repeat this for each  
value to find the largest*

Can you change this into a 'min\_list' quickly?

# Computing New Collections

We're not constrained to aggregation, but can transform existing collections into new collections.

Example: reverse the order of items in a list.

```
def reverse(list):  
    newlist = []  
    for item in list:  
        newlist = [item] + newlist  
    return newlist  
  
print(reverse([1,2,3,4,5]))
```

# Computing New Collections

We're not constrained to aggregation, but can transform existing collections into new collections.

Example: reverse the order of items in a list.

```
def reverse(list):
```

```
    newlist = []
```

```
    for item in list:
```

```
        newlist = [item] + newlist
```

```
    return newlist
```

```
print(reverse([1,2,3,4,5]))
```

*Temporary variable we'll use to build the new list*

*prepend each item to new list successively*

# Associative Dictionaries

We often need to *associate* data items:  
e.g., names of people with attributes such as age.

**Dictionaries** are collections of **key/value pairs**,  
each relating a key (e.g., name) to a value (age).

Now we can find values by *key* instead of *position*:

```
ages = {  
    "Alice":15, "Bob":30, "Charlie":68}  
  
print("Bob's age is", ages["Bob"])
```

# Loops over Dictionaries

We can use 'for' to loop over all the keys present in a dictionary and do something with their values.

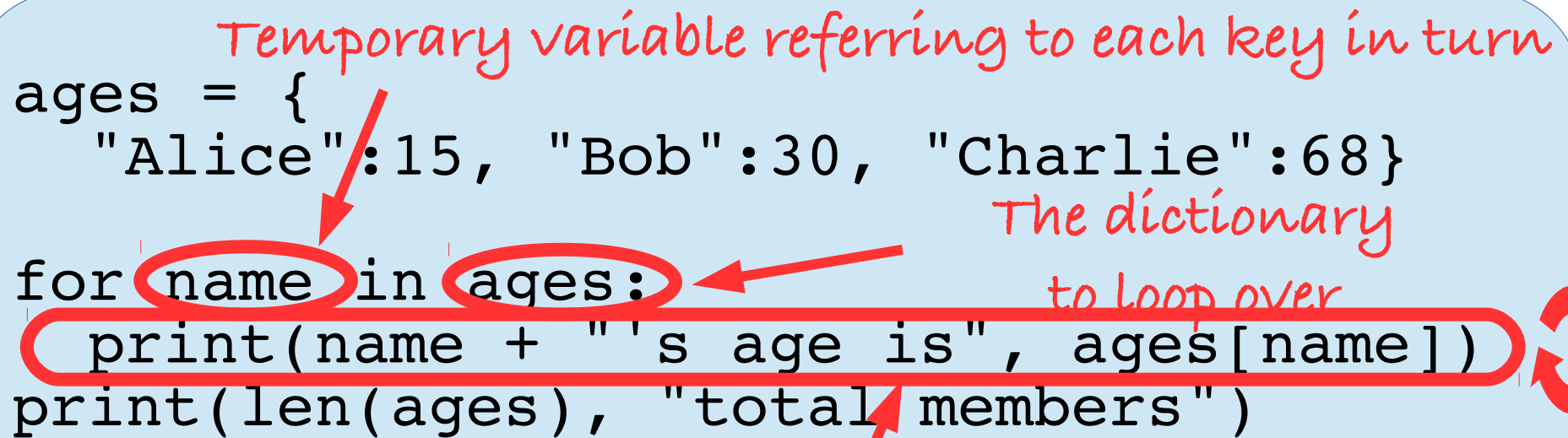
Example: print name and age of each person

```
ages = {  
    "Alice":15, "Bob":30, "Charlie":68}  
  
for name in ages:  
    print(name + "'s age is", ages[name])  
print(len(ages), "total members")
```

# Loops over Dictionaries

We can use 'for' to loop over all the keys present in a dictionary and do something with their values.

Example: print name and age of each person



```
ages = {  
    "Alice":15, "Bob":30, "Charlie":68}  
  
for name in ages:  
    print(name + "'s age is", ages[name])  
print(len(ages), "total members")
```

*Temporary variable referring to each key in turn*

*The dictionary to loop over*

*The code to run once for each key in the dictionary*

The code is annotated with red circles and arrows. A red circle around 'name' has an arrow pointing to it from the text 'Temporary variable referring to each key in turn'. A red circle around 'ages' has an arrow pointing to it from the text 'The dictionary to loop over'. A red circle around the entire loop body (the two print statements) has an arrow pointing to it from the text 'The code to run once for each key in the dictionary'. A large red circular arrow is on the right side of the code block.

# Example: Pseudonymizer

As with lists, we can transform whole dictionaries.

- Example: replace names with pseudonyms<sup>†</sup>

```
def pseudonymize(dict):  
    newdict = {}  
    nextnym = 1  
    for name in dict:  
        nym = "user" + str(nextnym)  
        newdict[nym] = dict[name]  
        nextnym = nextnym + 1  
    return newdict  
  
ages = {  
    "Alice":15, "Bob":30, "Charlie":68}  
print(pseudonymize(ages))
```

<sup>†</sup> A rather naive way to protect privacy of course, as we'll explore later



# Divide-and-Conquer via Recursion

We often want to break a large, hard problem into smaller, easier subproblems, *using the same code* to process the main problem and its subproblems.

We do this via **recursion**: a function calling itself.

Example:  $\text{factorial}(n)$  multiplies all integers  $1 \dots n$ .

# Example: Recursive Factorial

Example:  $\text{factorial}(n)$  multiplies all integers  $1 \dots n$ .

- $\text{factorial}(3)$  is  $1 \times 2 \times 3 = 6$
- $\text{factorial}(5)$  is  $1 \times 2 \times 3 \times 4 \times 5 = 120$

To compute  $\text{factorial}(n)$ , let's first solve the smaller problem of  $\text{factorial}(n-1)$ , then just multiply by  $n$ :

```
def factorial(n):  
    if n > 1:  
        return n * factorial(n-1)  
    return 1  
  
print(factorial(3), factorial(5))
```

# Example: Recursive Factorial

Example:  $\text{factorial}(n)$  multiplies all integers  $1 \dots n$ .

- $\text{factorial}(3)$  is  $1 \times 2 \times 3 = 6$
- $\text{factorial}(5)$  is  $1 \times 2 \times 3 \times 4 \times 5 = 120$

To compute  $\text{factorial}(n)$ , let's first solve the smaller problem of  $\text{factorial}(n-1)$ , then just multiply by  $n$ :

```
def factorial(n):  
    if n > 1:  
        return n * factorial(n-1)  
    return 1
```

*Recursive function call*

```
print(factorial(3), factorial(5))
```

# How Recursive Functions Work

The computer keeps track of multiple *partially* executed instances of factorial function at once.

**Call factorial(3)** from main program:  $n = 3$

Is  $n$  (3) greater than 1? Yes.

**Call factorial(2)** from factorial(3):  $n = 2$

Is  $n$  (2) greater than 1? Yes.

**Call factorial(1)** from factorial(2)

Is  $n$  (1) greater than 1? No.

Return 1.

Return 2 times factorial(1)'s result (1)

Result of factorial(2) = 2

Return 3 times factorial(2)'s result (2)

Result of factorial(3) = 6

# Pitfalls of Recursion (1)

**Infinite recursion:** what happens if you forget the condition that stops subdividing the problem.

Example: factorial( $n$ ) without the 'if' condition:

```
def factorial(n):  
    return n * factorial(n-1)  
  
print(factorial(3), factorial(5))
```

What happens?

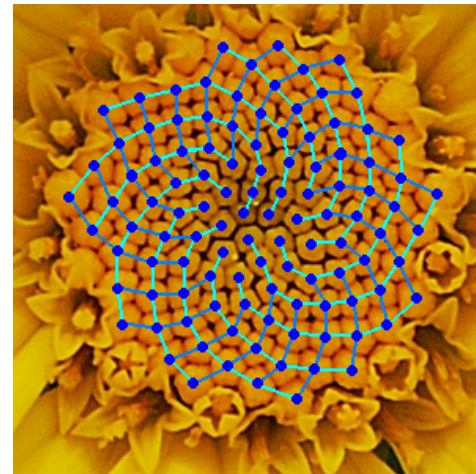
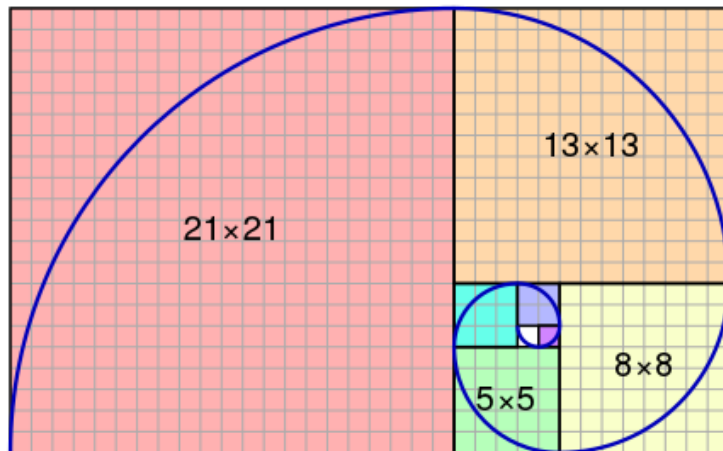
# Example: Fibonacci Numbers

Fibonacci numbers are extremely simple but fundamental in both mathematics and nature:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

Notice that all but the first two are simply the *sum of the previous two numbers* in the series.

They relate to golden spirals and flower petals...



# Fibonacci Numbers via Recursion

We can compute  $\text{fibonacci}(n)$  just like  $\text{factorial}(n)$ , breaking it down into the smaller sub-problems of computing  $\text{fibonacci}(n-1)$  and  $\text{fibonacci}(n-2)$ :

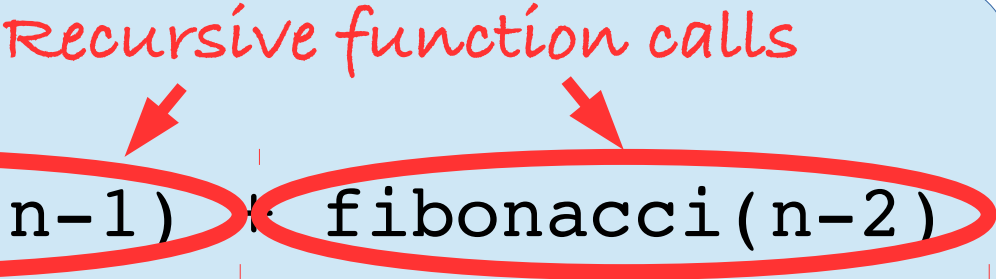
```
def fibonacci(n):  
    if n > 2:  
        return fibonacci(n-1) + fibonacci(n-2)  
    return 1  
  
for n in range(1, 10):  
    print(fibonacci(n))
```

# Fibonacci Numbers via Recursion

We can compute  $\text{fibonacci}(n)$  just like  $\text{factorial}(n)$ , breaking it down into the smaller sub-problems of computing  $\text{fibonacci}(n-1)$  and  $\text{fibonacci}(n-2)$ :

```
def fibonacci(n):  
    if n > 2:  
        return fibonacci(n-1) + fibonacci(n-2)  
    return 1  
  
for n in range(1, 10):  
    print(fibonacci(n))
```

*Recursive function calls*





# Pitfalls of Recursion (2)

What happens if you try to use the above program to print all the Fibonacci numbers up to 100?

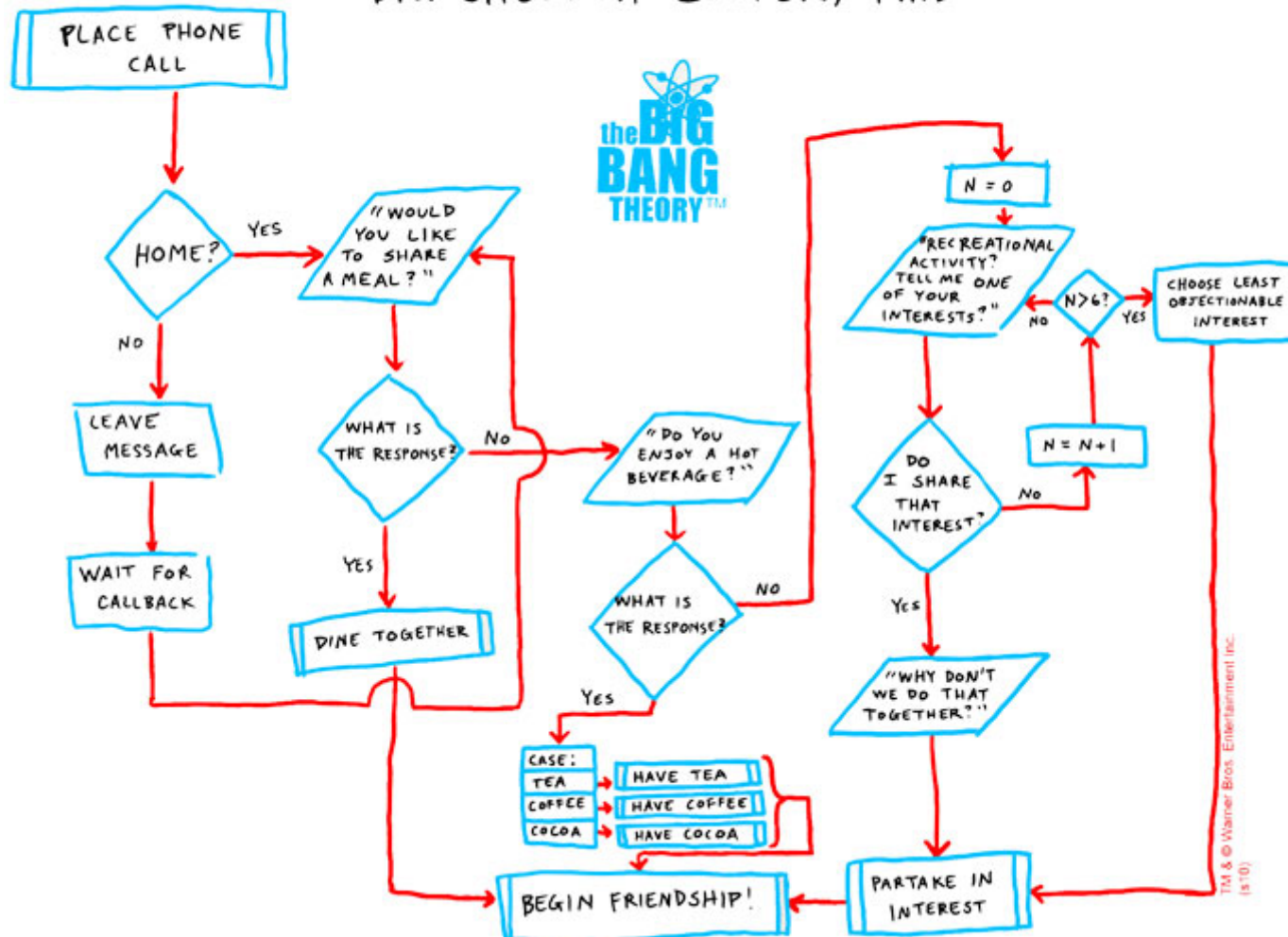
Try it!

Why do you think it behaves this way?

# Part 2: Algorithms and Complexity

## THE FRIENDSHIP ALGORITHM

DR. SHELDON COOPER, Ph.D



# Algorithmic Complexity

Just because an algorithm works *in principle* doesn't necessarily make it *practical* to use.

**Algorithmic complexity** is the analysis of the *performance costs* of running an algorithm on an input data set or problem of a given size.

- The focus is not on predicting costs precisely, but on *categorizing* algorithms by *how quickly* their costs grow as the problem size grows.
- Helps us distinguish algorithms that “*may be*” *practical* from those that are *wildly impractical*.

# Kinds of Complexity Costs

We care about multiple kinds of costs, e.g.:

- **Computation:** how many “steps” or units of work a processor must perform to complete it
- **Storage:** how many bytes of memory or disk space it requires to store intermediate results.
- **Communication:** how many messages or bytes a distributed algorithm must transmit.

We will focus on computation, but the fundamental principles are the same for analyzing other costs.

# Complexity Example

Compare the recursive factorial and fibonacci algorithms presented above. Suppose `factorial(1)` and `fibonacci(1)` each take 1 second to execute. How long will they take with an input  $n$  of 100?

- `factorial(100)` can be expected to take “on the order of” around 100 seconds to complete.
- `fibonacci(100)` can be expected to take “on the order of” around *35 million years* to complete.

They both look (and work) quite similarly but have vastly different computational complexity. Why?

# Analyzing factorial vs fibonacci

Recall that factorial( $n$ ) contains *one* recursive function call, whereas fibonacci( $n$ ) contains *two*:

```
def factorial(n):  
    if n > 1:  
        return n * factorial(n-1)  
    return 1
```

```
def fibonacci(n):  
    if n > 2:  
        return fibonacci(n-1) + fibonacci(n-2)  
    return 1
```

How many calls to each occur when  $n = 100$ ?

# Analyzing factorial vs fibonacci

Calling factorial(100)

→ 1 call to factorial(99)

Calling factorial(99)

→ 1 call to factorial(98)

...

Calling fibonacci(100)

→ 1 call to fibonacci(99)

→ 1 call to fibonacci(98)

Calling fibonacci(99)

→ 1 call to fibonacci(98)

→ 1 call to fibonacci(97)

Notice that *one* call to fibonacci(100) causes *two* redundant calls to fibonacci(98).

Each of those makes two calls to fibonacci(96),  
or  $2^2=4$  redundant calls to fibonacci(96) in total.

...until fibonacci(2) ends up getting called  $2^{50}$  times!

# Linear vs Exponential Complexity

Recursive factorial has **linear** or complexity: execution time increases proportionally with  $n$ .

- Mathematically, there is some constant  $c$  such that execution time  $T = c \times n$ .

Recursive fibonacci has **exponential** complexity: execution time increases exponentially with  $n$ .

- Mathematically, there is some constant  $c$  such that execution time  $T = c \times 2^n$

Recursive factorial is **efficiently computable**, whereas recursive fibonacci is definitely not!



# Complexity and Big-Oh Notation

Since algorithmic complexity focuses on how execution costs *grow* with the problem size ( $n$ ), the specific constant  $c$  doesn't really matter.

- $\text{factorial}(n)$  running on a faster CPU will have a smaller constant  $c$ , but will *grow* the same way.

Computer scientists use “Big-O” notation to hide these irrelevant constants and focus on growth.

- $O(n)$  means there is a  $c$  such that  $T = c \times n$ .
- $O(n^2)$  means there is a  $c$  such that  $T = c \times n^2$ .
- $O(2^n)$  means there is a  $c$  such that  $T = c \times 2^n$ .

# Polynomial Complexity

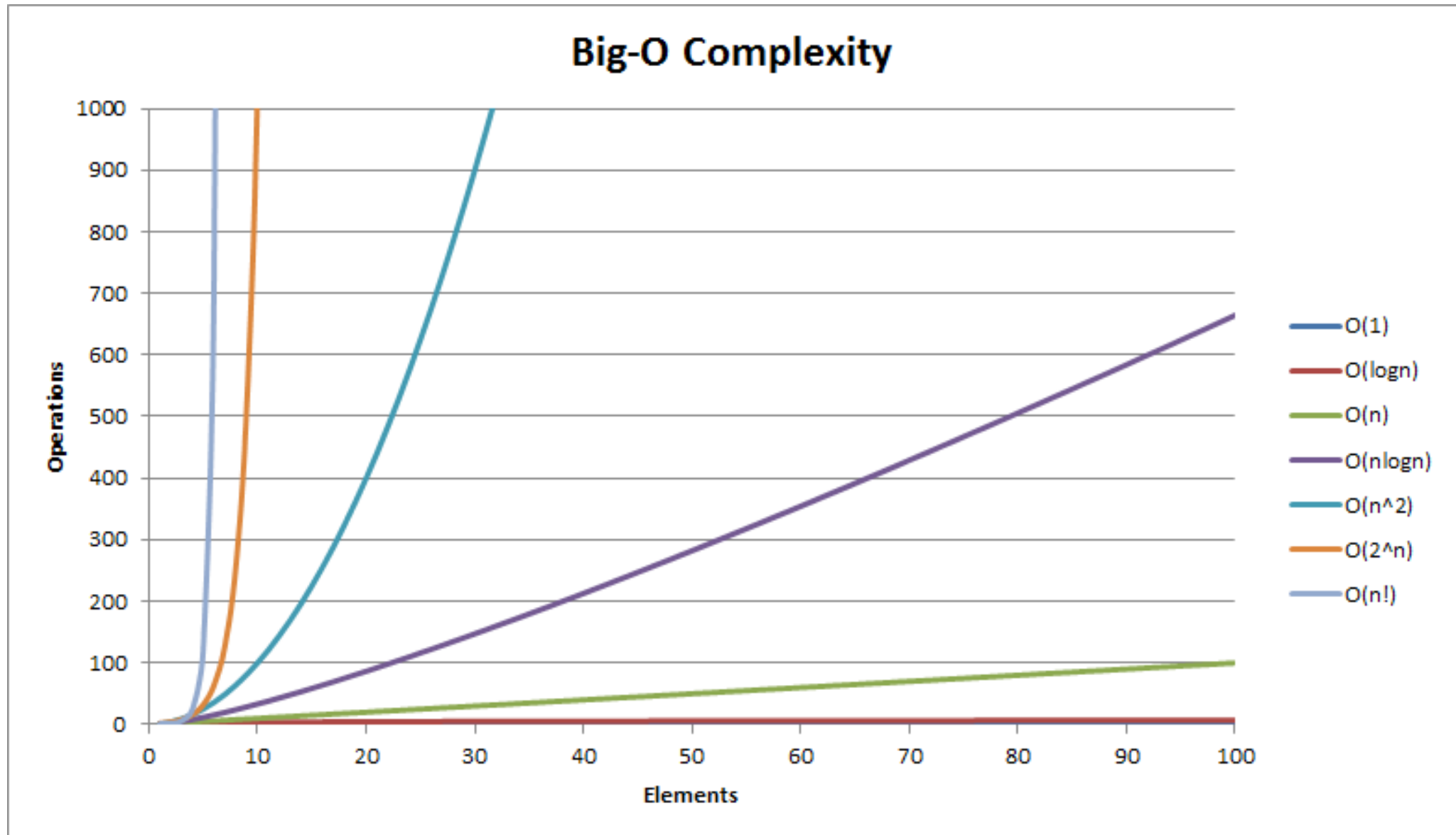
Theoretical computer scientists consider a problem to be *efficiently computable* if there is a **polynomial-time algorithm** to compute it.

This means there is some constant power  $p$  such that the algorithm runs in time  $O(n^p)$ .

- **Constant-time** execution is  $O(1)$ , or  $p = 0$ .
- **Linear-time** execution is  $O(n)$ , or  $p = 1$ .
- **Quadratic-time** execution is  $O(n^2)$ , or  $p = 2$ .

The polynomial power  $p$  cannot depend on  $n$ , otherwise runtime would be exponential in  $n$ !

# Complexity Classes Illustrated



# Search Algorithms

Let's analyze the complexity of some simple algorithms for *finding something in a collection*.

Example: given a Python list of group members, determine if a particular name is in the group.

We will look at three classes of algorithms:

- Linear-time,  $O(n)$ : slow for search algorithms
- Logarithmic-time,  $O(\log n)$ : much better
- Constant-time,  $O(1)$ : even better!

# Linear-Scan Search Algorithm

The simplest approach: simply scan the list.

```
def member(name, group):  
    for member in group:  
        if member == name:  
            return name + " is a member"  
    return name + " isn't a member"  
  
group = ["Alice", "Bob", "Charlie"]  
print(member("Alice", group))  
print(member("Dave", group))
```

Works fine, but takes **linear time**,  $O(n)$ , because we may test against all  $n$  members of the group.

# Logarithmic-time Binary Search

Significantly faster searches generally require that the data be *organized* in some way, e.g., sorted.

If so, we can use more-or-less the same algorithm that we use manually searching a dictionary:

- 1) Open to a page in (approximately) the middle.
- 2) Use alphabetical ordering to check whether the desired word is before, on, or after this page.
- 3) If not found on this page, repeat from step 1, restricting attention to pages before/after this.

# Example Binary Search in Python

A bit more complex, but recursion helps...

```
def member(name, list):  
    if len(list) == 0:  
        return name + " isn't a member"  
    mid = len(list) // 2  
    if list[mid] == name:  
        return name + " is a member"  
    elif list[mid] > name:  
        return member(name, list[0:mid])  
    else:  
        return member(name, list[mid+1:])
```

Divide-and-conquer!

# How big of a win is binary search?

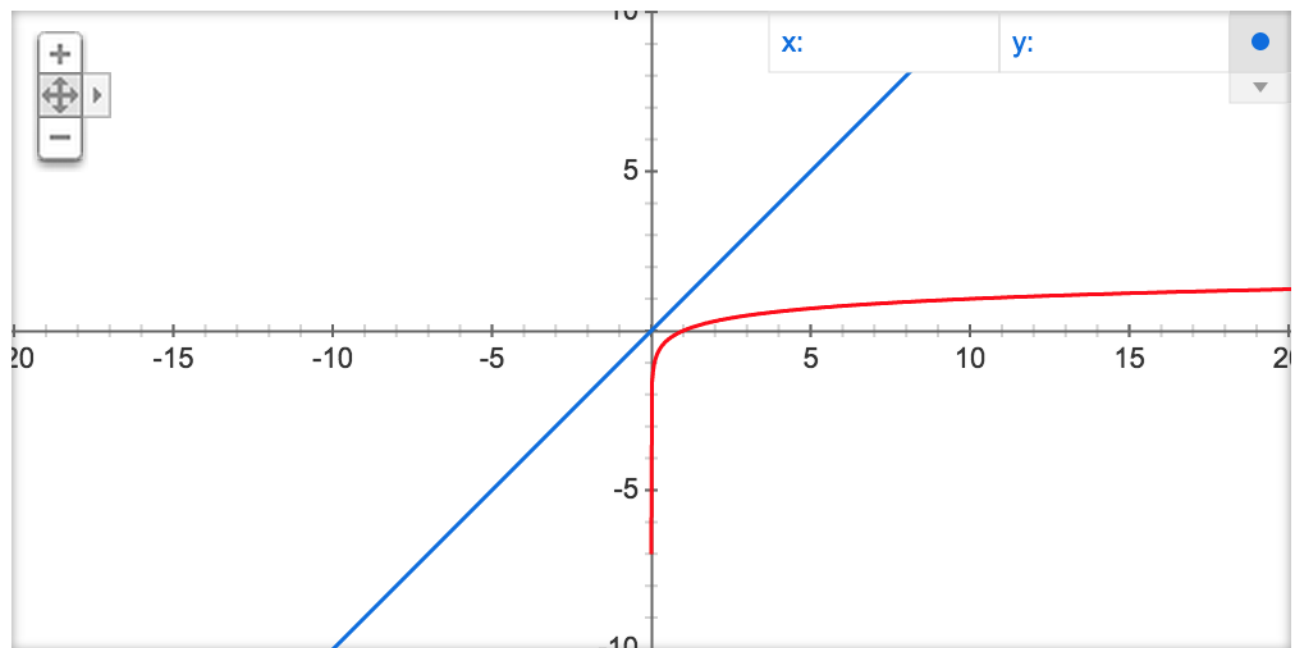
On short lists (e.g.,  $n = 10$ ), you won't notice.

But as  $n$  gets large, the difference becomes huge.

**Linear scan:**  
doubling  $n$   
doubles time.

**Binary search:**  
doubling  $n$   
adds *one* step!

Graph for  $x$ ,  $\log(x)$





# Can we do even better?

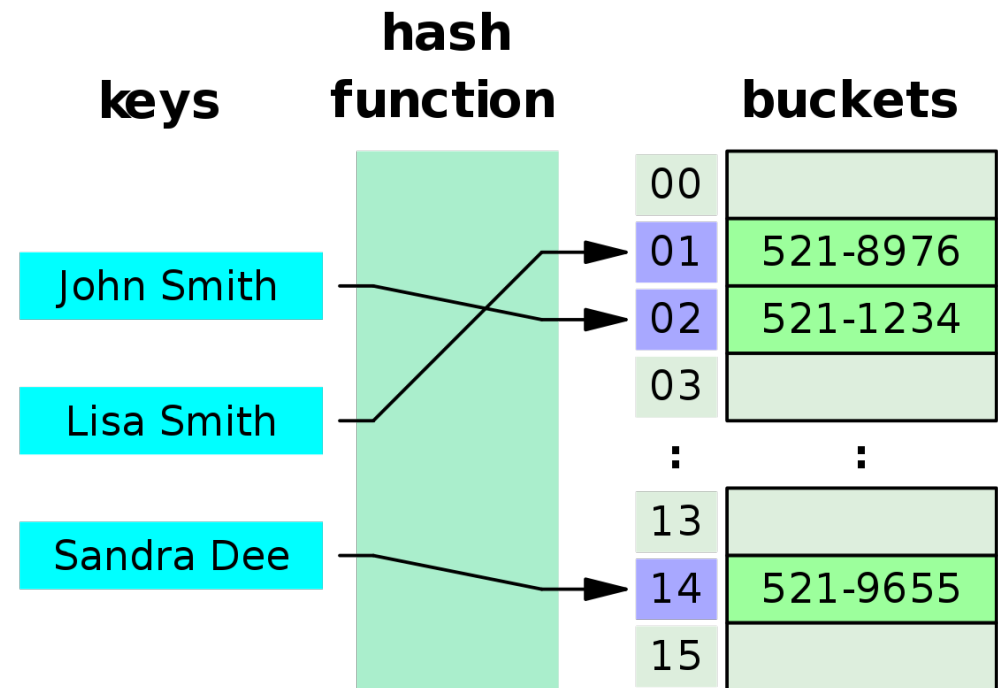
Yes, with some technical issues and caveats:

**Hash tables** can provide constant-time  $O(1)$  cost.

1) Use a *hash function* to map names (keys) to indexes in a table

2) Look in that one hash table entry

3) Hope that names don't [often] “collide” by hashing to the same entry!



# Sorting Algorithms

We can search efficiently in well-organized data, but what are the costs of *organizing* it?

**Sorting algorithms** take a list of data items and put them in order (e.g., numeric or lexicographic).

- Slow sorting algorithms are simple and easy; faster sorting algorithms are bit more complex.
- We will use this website to explore visually how classic sorting algorithms work:

<https://visualgo.net/en>

# Selection Sort: $O(n^2)$

A **selection sort** steps over the list, selecting the next-smallest element that fits into each position.

- 1) Find the smallest item, move it to position 1
- 2) Find second-smallest item, move to position 2
- 3) ...

But each step requires *searching* the unsorted part of the list for the smallest remaining item, which takes up to  $n$  computation operations.

- $n$  selection steps  $\times$   $n$  search steps each =  $O(n^2)$

# Example Selection Sort in Python

```
def sort(list):  
    for i in range(len(list)):  
        smallest = i  
        for j in range(i+1, len(list)):  
            if list[j] < list[smallest]:  
                smallest = j  
        temp = list[i]  
        list[i] = list[smallest]  
        list[smallest] = temp
```

```
list = [5, 7, 3, 9, 1, 6]  
sort(list)  
print(list)
```

# Selection Sort Illustrated



$n$  operations



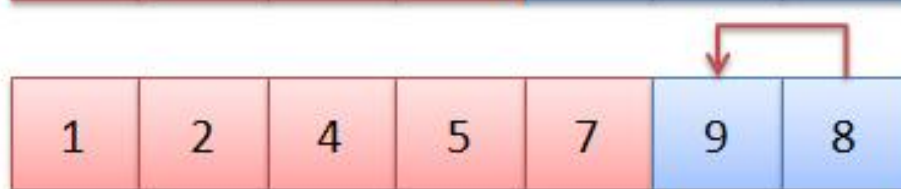
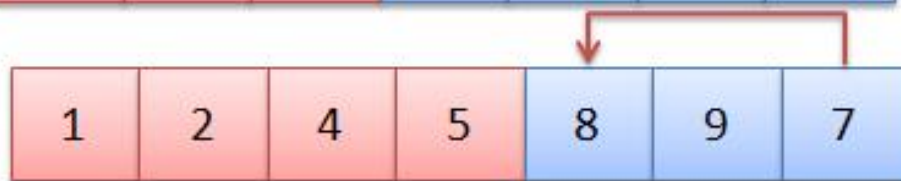
$n-1$  operations



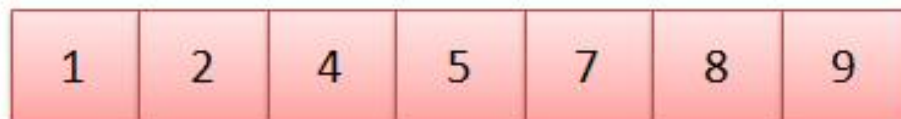
$n-2$  operations



...



$1$  operation



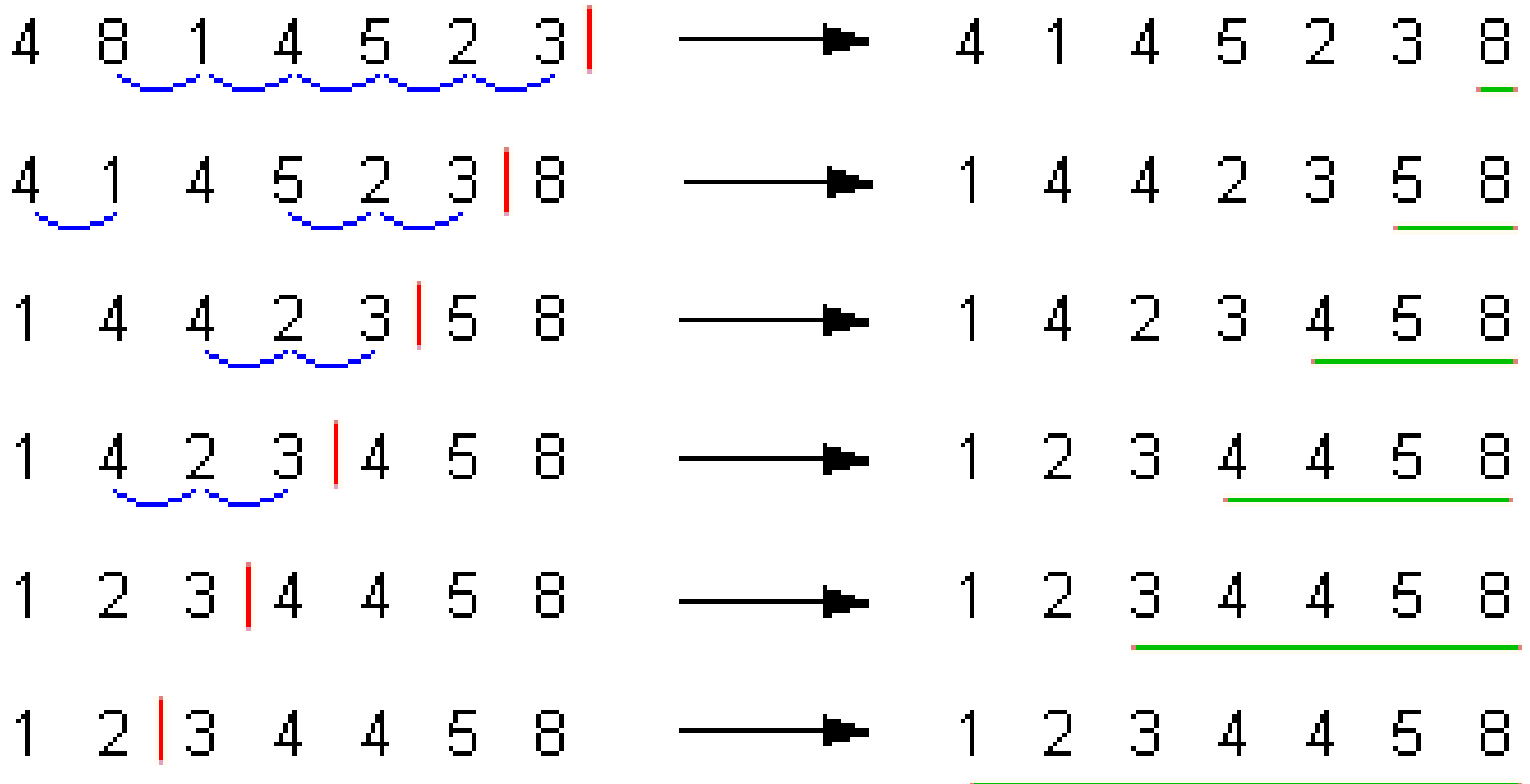
done!

# Bubble Sort: $O(n^2)$

A **bubble sort** simply traverses the list repeatedly, swapping all adjacent out-of-order *pairs* it finds, until it finds no further out-of-order pairs to swap.

```
def sort(list):  
    out_of_order = True  
    while out_of_order:  
        out_of_order = False  
        for i in range(len(list)-1):  
            if list[i] > list[i+1]:  
                temp = list[i]  
                list[i] = list[i+1]  
                list[i+1] = temp  
                out_of_order = True
```

# Bubble Sort Illustrated



# Bubble Sort Complexity Analysis

Bubble sort has computational complexity  $O(n^2)$ :

- It may require up to  $n$  passes over the list.
- Each pass requires  $n-1$  comparison steps.
- $O(n)$  passes  $\times O(n)$  comparisons =  $O(n^2)$

In general, sorting algorithms that depend on strictly linear traversals of the data are  $O(n^2)$ .

Can we do better? Yes: divide and conquer!



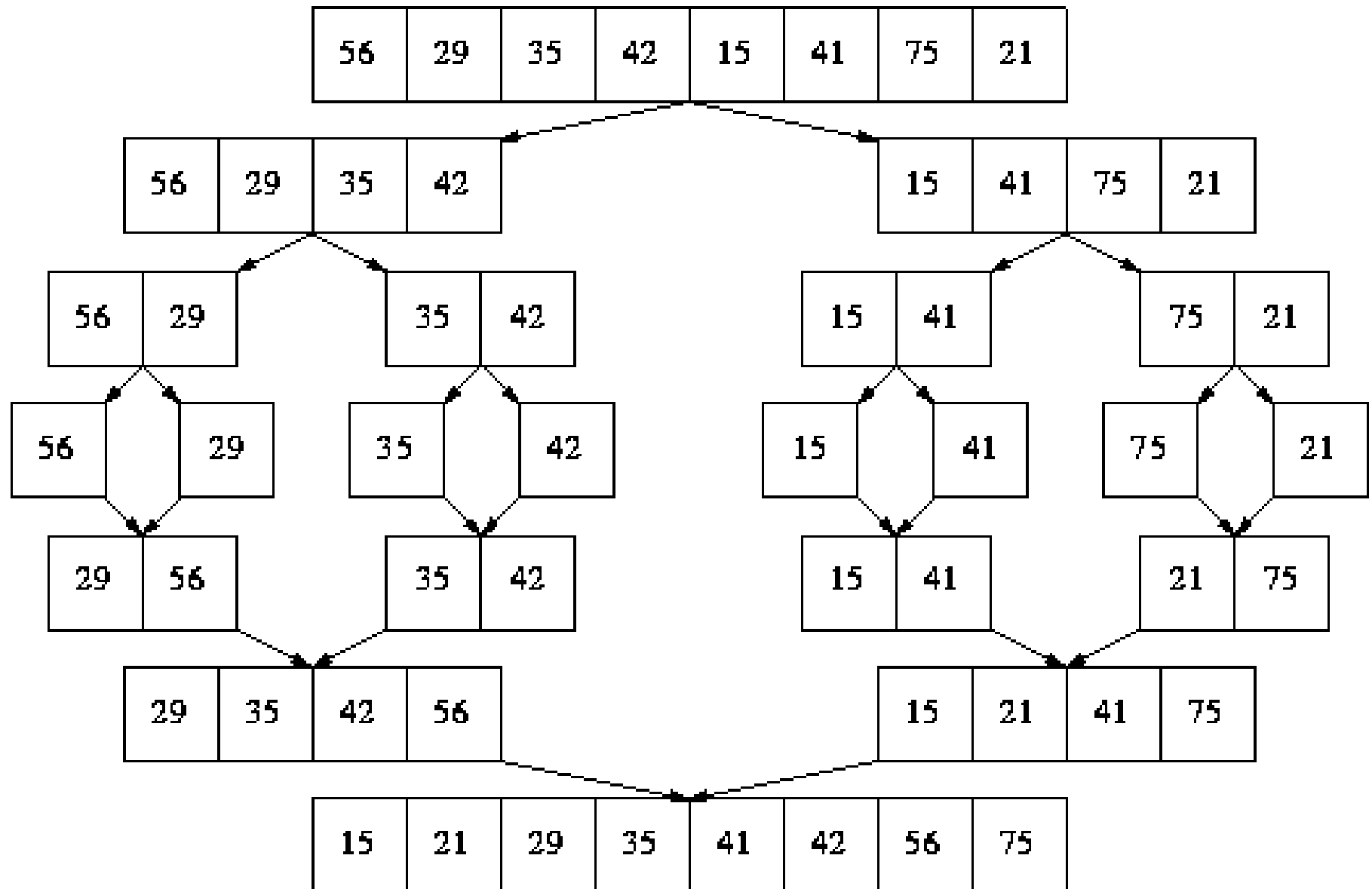
# Merge Sort: $O(n \log n)$

A **merge sort** divides a long list in half, *recursively* sorts the two halves, then *merges* the results.

In particular, to sort a list:

- 1) If the list has only one item, just return it.
- 2) Divide the list into “left” and “right” halves.
- 3) Recursively call merge-sort to sort each half.
- 4) Merge the two sorted halves into one list.

# Merge Sort Illustrated



# Example Merge Sort in Python

```
def sort(list):
    if len(list) <= 1:
        return list
    mid = len(list) // 2
    l = sort(list[0:mid])
    r = sort(list[mid:])
    merged = []
    while len(l) > 0 or len(r) > 0:
        if len(l) > 0 and
           (len(r) == 0 or l[0] < r[0]):
            merged.append(l[0])
            l = l[1:]
        else:
            merged.append(r[0])
            r = r[1:]
    return merged
```

# Example Merge Sort in Python

```
def sort(list):  
    if len(list) <= 1:  
        return list  
    mid = len(list) // 2  
    l = sort(list[0:mid])  
    r = sort(list[mid:])  
    merged = []  
    while len(l) > 0 or len(r) > 0:  
        if len(l) > 0 and  
            (len(r) == 0 or l[0] < r[0]):  
            merged.append(l[0])  
            l = l[1:]  
        else:  
            merged.append(r[0])  
            r = r[1:]  
    return merged
```

*Subdivide the problem*

*Sort each sub-list*

*Merge the sub-lists*

# Merge Sort Complexity Analysis

Three key observations:

- Because each recursive divide-and-conquer level divides the list in half, we need at most  $O(\log n)$  levels of recursion.
- Merging two lists of length  $n/2$  requires only a *single pass* and thus operates in  $O(n)$  time.
  - Repeatedly pick the *first item* from the left sub-list or the *first item* from the right, whichever is smaller.
- The total “merging work” at each level is  $O(n)$ .

$O(\log n)$  levels  $\times$   $O(n)$  work per level =  $O(n \log n)$

# Quick Sort: $O(n \log n)$ *expected!*

**Quick sort** is a classic *randomized algorithm*

- Simpler and *almost always* faster than most known deterministic sorting algorithms

Also uses recursion to divide-and-conquer:

- 1) Pick a *random* list element as the *pivot*.
- 2) Divide the list into two *unsorted* sub-lists, containing items lower/higher than the pivot.
- 3) Recursively quick-sort each of the sub-lists.
- 4) Concatenate the sub-lists to form the result.

# Quick Sort Complexity Analysis

Key observations:

- Quick Sort will require at most  $O(\log n)$  levels of recursion *with high probability*.
  - We *could* consistently pick “bad” pivots producing unbalanced sub-lists, but it’s *exponentially* unlikely
- Dividing the big list into two sub-lists is  $O(n)$ .
  - Compare each item with pivot, put in “left” or “right”
- Concatenating sorted sub-lists is  $O(n)$ .

$O(\log n)$  levels  $\times O(n)$  work per level =  $O(n \log n)$

# Matrix Multiplication

A huge number of data processing applications, especially in graphics and signal processing, depend heavily on **matrix multiplication**.

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}$$

$A \qquad B \qquad C$

In short: each *cell* of the result matrix  $C$  is the *inner product* of a row in  $A$  and a column in  $B$ .



# Matrix Multiplication Complexity

The simple, obvious algorithm is  $O(n^3)$  because:

- There are  $O(n^2)$  cells in the result to be filled.
- Computing each result cell requires multiplying  $n$  cells of a row in  $A$  by  $n$  cells of a column in  $B$ .

Thus, matrix multiplication is fundamentally more costly than searching or sorting as  $n$  grows large.

- But still polynomial time:  $n^3$  is a polynomial.

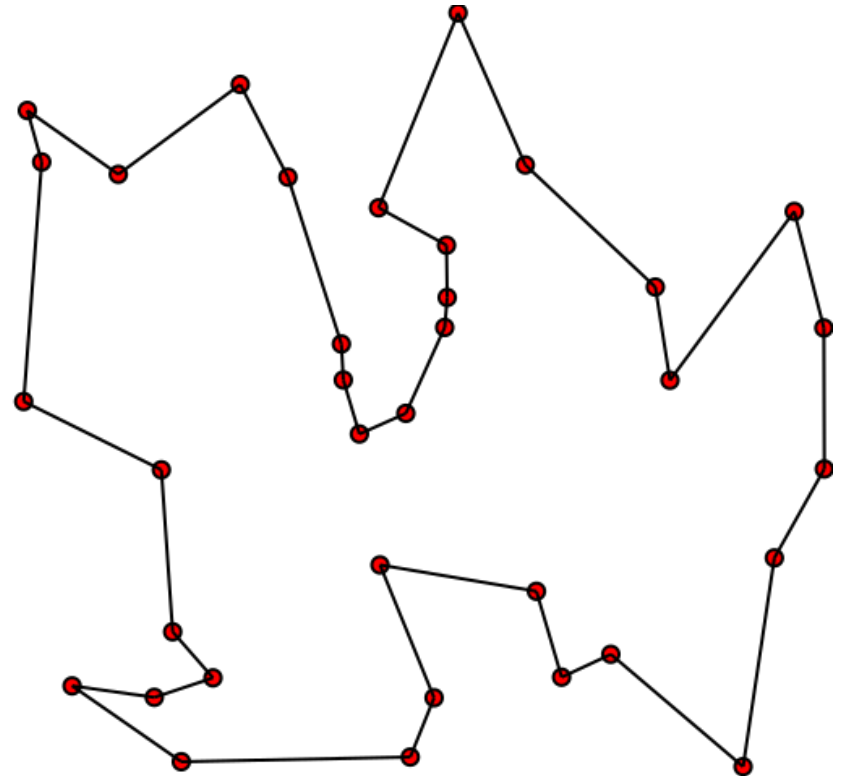
With fancier, more complex algorithms, can be reduced to  $O(n^{2.3737})$  and maybe even better...

# Exponential-Time Algorithms

For many problems, there is no known algorithm guaranteed to complete in polynomial time

- Example: the **Traveling Salesman Problem**

Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?



# Why are Algorithms Exponential?

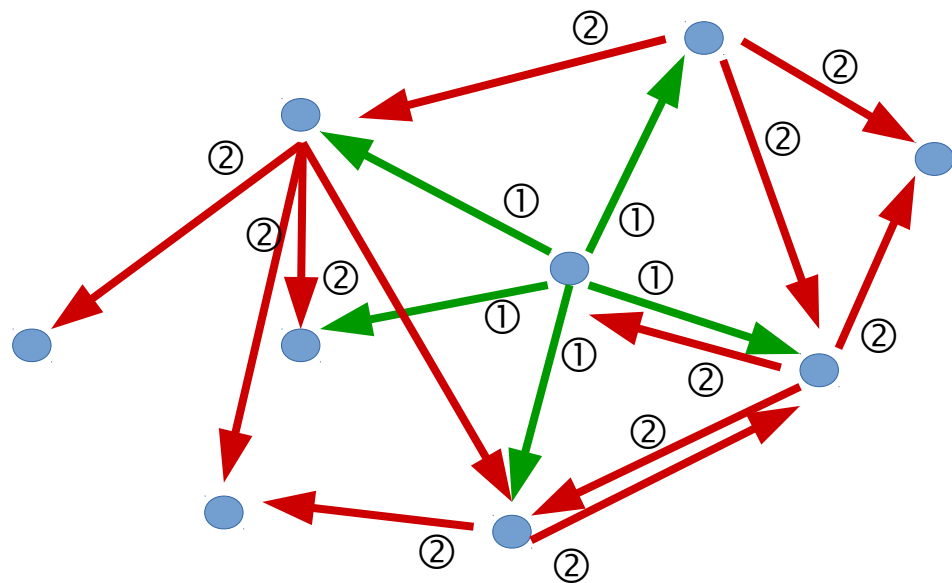
Usually: because we have to search a space of possible answers that explodes *multiplicatively* after each step of progress in the search.

Traveling salesman example: *which city next?*

- No apparent way much better than *try them all*

After step...

- 1)  $n$  choices
  - 2)  $n^2$  choices
  - 3)  $n^3$  choices
- etc...



# Recursive Fibonacci Revisited

The recursive Fibonacci algorithm from before takes exponential time in input  $n$ . Essential?

**No:** that was because of a *bad implementation*.

- It's surprisingly easy to get exponentially bad performance accidentally from an algorithm bug

We can compute Fibonacci numbers quickly just by ensuring that we compute each one only once.

- Example: initialize a list to  $[1, 1]$ , then successively extend until length  $n$ .

But there are also *real* exponential-time problems

# NP-complete and NP-hard problems

For a large class of **NP-complete** problems, it seems extremely unlikely that a polynomial-time algorithm exists, but *we can't (yet) prove it*.

- NP means “nondeterministic polynomial-time”
  - Simplified: answer is hard to find but easy to check
- One of 6 open theory problems with a \$1 million prize from the [Clay Mathematics Institute](#)

**NP-hard** problems are at least as hard as NP-complete problems and maybe even harder

# Session 2 Wrap-Up

Part 1: a crash course in Python programming

- Data types: integers, strings, lists, dictionaries
- Operators: integer, string, relational, logical
- Abstracting with variables and functions
- Flow control: conditionals and loops
- Recursion: functions calling themselves

# Session 2 Wrap-Up

## Part 2: basics of algorithms and complexity

- How computation costs grow with problem size
  - Ignoring constants with Big-O notation
- “Efficiently computable” complexity classes:
  - Constant time:  $O(1)$ : e.g., hash table lookup
  - Logarithmic time:  $O(\log n)$ : e.g., binary search
  - Linear time:  $O(n)$ : e.g., linear search
  - Quasilinear time:  $O(n \log n)$ : merge sort, quick sort
  - Polynomial time:  $O(n^p)$ : e.g., matrix multiply
- Exponential-time complexity:  $O(2^n)$ : *impractical!*