

# Introduction to Computer Science

Rahul Narain

COL100  
I Semester, 2021-22

# Welcome

This course is about the fundamentals of computer science.

But what *is* computer science?

Is it about *computers*? Not really!

*“Computer science is no more about computers than astronomy is about telescopes.”*

—CS folklore

Is it about *programming*? Not really.

Computer science is about *computation*: processes for finding solutions to problems.

Given a problem, can a solution be found? How much time / space / resources / ... does it take?

## Examples

Given two  $n$ -digit natural numbers, find their product.

$$\begin{array}{r} 121 \\ \times 121 \\ \hline 121 \\ 242 \\ 121 \\ \hline 14641 \end{array}$$

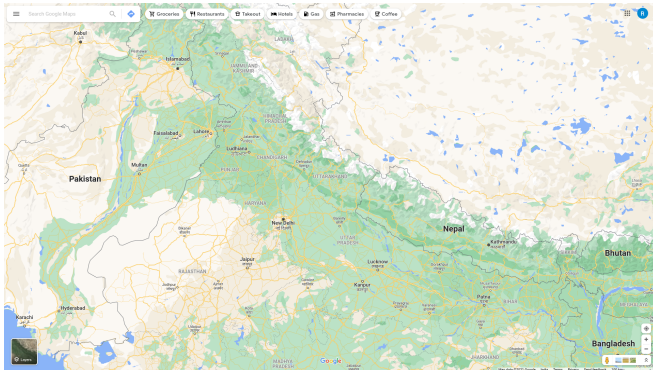
Does this procedure always work? *Why?*

How many single-digit operations does it require?  $O(n^2)$

Is it possible to do it with fewer operations? (Surprisingly, yes!)

# Examples

Given a road map with  $n$  cities, what is the shortest route from one city to another? What is the shortest round trip covering all  $n$  cities?



How many steps do these problems take?

No *efficient* solution to the second problem!

# Examples

Given infinite supplies of  $n$  bricks with a word on the top and a word on the bottom, is it possible to arrange a sequence of them to spell the same word on both sides?

a	ab	bba
baa	aa	bb

1

2

3



This problem is *undecidable*! No systematic way of solving *all* possible instances.

My point: None of these problems required talking about *computers* or about *programming*.

# Computing devices and levels of abstraction

Adding, subtracting, multiplying, dividing  $n$ -digit numbers: Can be done on many different devices.

- ▶ Pencil and paper
- ▶ Abacus
- ▶ Calculator
- ▶ ...

Underlying machinery is totally different! But at a higher level of abstraction, all are equivalent: devices for computing arithmetic.

All are equally sufficient for performing higher-level calculations.

# Computing devices and levels of abstraction

Given a device for computing arithmetic, we can build more abstractions on top.

**Example:** Using only numbers, how can we represent *text*, e.g. “Hello”?

Choose an *encoding* of characters as numbers:

...	64	65	66	67	...	90	91	...	97	98	99	...
...	@	A	B	C	...	Z	[	...	a	b	c	...

Hello = 72, 101, 108, 108, 111

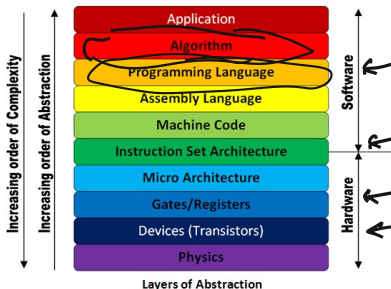
Now you can do text operations using only arithmetic, e.g.

- ▶ Is a given character (encoded as a number) a letter?
- ▶ Capitalize every letter of a given word

# Computer architecture

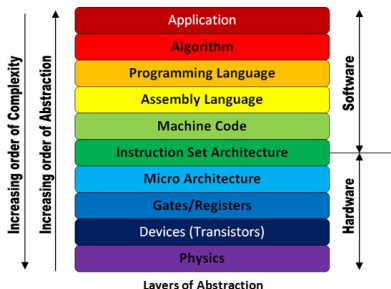
We can build highly sophisticated computing machinery (a.k.a. “computers”) using multiple levels of abstraction.

Each level offers some types of data representation and some built-in operations





# Computer architecture



Details of lower levels (mostly) don't matter, can be replaced by a different "implementation"!

Abacus, calculator, human with pen and paper, water in pipes(!), ...



We will be working at the programming language level, but may see examples of other levels, e.g. how arithmetic is done using 0s and 1s.

# Thinking like a computer scientist

We have some computing device. It offers some data representations, and some operations. In other words, it provides some *model of computation*.

We want to solve some given problem (or rather, a whole class of problems).

We will have to *encode* the problem information into the available data types.

Then, we will have to design a systematic way of finding the solution using the available operations, that is, an *algorithm*.

## Example: Primality testing

*Data:* integers. *Operations:* arithmetic.

*Problem:* Given an integer  $n > 1$ , is it prime?

## Example: Primality testing

*Data:* integers. *Operations:* arithmetic.

*Problem:* Given an integer  $n > 1$ , is it prime?

**Algorithm:**

For each  $i$  in  $2, 3, 4, \dots, n-1$ ,

If  $n \bmod i = 0$ , answer "no" and stop.

Otherwise, carry on with the next  $i$ .

→ If you've finished going through all  $i$ 's, answer "yes".

The same algorithm can be expressed in many other ways:

```
def prime(n):  
    for i in range(2, n):  
        if n % i == 0:  
            return False  
    return True
```

python

$$\left\{ \begin{array}{l} p : \mathbb{N} \rightarrow \{0, 1\} \\ p(n) = q(n, 2) \\ q(n, i) = \begin{cases} 1 & \text{if } i = n, \\ 0 & \text{if } n \bmod i = 0, \\ q(n, i + 1) & \text{otherwise} \end{cases} \end{array} \right.$$

## Example: Primality testing

For each  $i$  in 2, 3, 4, ...,  $n - 1$ ,

    If  $n \bmod i = 0$ , answer “no” and stop.

    Otherwise, carry on with next  $i$ .

If you’ve finished going through all  $i$ ’s, answer “yes”.

An algorithm must be possible to carry out mechanically, requiring no clever extra steps outside the allowed operations.

All the cleverness goes into designing the algorithm in the first place!

*Of course, the usual questions apply:* How to prove that this algorithm answers “yes” if and only if  $n$  is prime? How many arithmetic operations does it require (at worst)? Can we do (significantly) better?

# Example: Navigation app

*Data:* numbers, sequences, matrices.

*Problem:* What is the length of the shortest route between two cities?

... First of all, how to represent a road map in the first place?



## Example: Navigation app

Give each city a number, e.g. Delhi = 1, Jaipur = 2, Lucknow = 3, ...

Encode network of roads...

- ▶ as a sequence of triples (city  $i$ , city  $j$ , length of road), e.g.  
 $\langle (1, 2, 281), (1, 4, 243), (2, 4, 238), (3, 4, 330), \dots \rangle$
- ▶ Or as an  $n \times n$  matrix whose  $(i, j)$ th entry is length of direct road between city  $i$  and city  $j$ , e.g.

$$\begin{bmatrix} 0 & 281 & \infty & 243 & \dots \\ 281 & 0 & \infty & 238 & \dots \\ \infty & \infty & 0 & 330 & \dots \\ 243 & 238 & 330 & 0 & \dots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

Now we may be able to solve the problem: Given two cities  $i$  and  $j$ , what is the length of the shortest route between them?

# Algorithms and programs

*Algorithm* = systematic procedure for solving problem in given model of computation.

Can be expressed in many ways. Not all can be “understood” by a computer. . .

*Programming language* = artificial language for expressing algorithms unambiguously (in a machine-executable model of computation)

Many different programming languages out there!

- . . . , C, C++, C#, Java, JavaScript, . . .
- . . . , Perl, PHP, Python, Ruby, . . .
- . . . , Scala, Go, Rust, Lisp, Scheme, . . .
- . . . , ML, OCaml, Haskell, Prolog, . . .

Which one should you learn?? *Doesn't matter.*



# Algorithms and programs

The programming language doesn't matter (a lot).

The real effort is in:

- ▶ designing an algorithm,
- ▶ proving that it always gives the correct results, and
- ▶ analyzing its efficiency (amount of time and space it takes).

After that, translating it into a programming language is relatively straightforward.

We will use *Python*, a beginner-friendly yet practically useful language.

We will *not* teach all the details of Python. You are expected to pick up the specifics by reading other resources on your own.

Any questions so far?

# Reminder of course logistics

Three instructors: Rahul Narain, Sanjiva Prasad, Preeti Ranjan Panda.

**Lectures** Mon, Thu 9:30–11am on Impartus (2101-COL100).

Slides posted afterwards on Moodle.

**Extra lectures** on Saturdays, once every two weeks. Details TBA.

**Lab sessions** on Teams (2101-COL100Lx where x is your group number).

Run by TAs. One day a week, 1-3pm or 3-5pm depending on the group.

**Outside-class discussion** on Piazza. Please participate actively, post any questions, discussions, ideas there!

# Course policies

## **Evaluation** (subject to change):

- ▶ Assignments (35%): one every two weeks, 6 in total
- ▶ Quizzes (15%): randomly during lectures
- ▶ Minor (20%)
- ▶ Major (30%)

To get a passing grade, students must:

- ▶ Submit at least 4 assignments and give demos, and
- ▶ Get at least 20% in assignments, and
- ▶ Get at least 20% in exams

**Late submissions:** 6 late-day tokens across all assignments. At most 2 can be used per assignment.

## Course policies

**Attendance** is required in all lectures and labs and will be tracked. Students who miss any session must document why.

**Plagiarism:** All students are expected to follow the highest ethical standards. Any instances of academic misconduct such as cheating *or helping others cheat* in exams, assignments, or quizzes will result in a severe penalty. This could include: a 0 in that test, an F grade in the entire course, or disciplinary action at the institute level. Allowing one's work to be available or visible to other students, whether deliberately or by negligence, will also be considered academic misconduct and treated as such.

# Python


www.pythonanywhere.com



Python3.8 console 22366572

 Share with others



```
Python 3.8.0 (default, Nov 14 2019, 22:29:45)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 2 + 2
4
>>> 
```

# Python

Display a simple message:

```
>>> print('Hello, world!')  
Hello, world!
```

Try some simple arithmetic:  $40 + 2$ ,  $43 - 1$ ,  $6 * 7$ ,  $84 / 2$

Division, quotient, remainder:  $19 / 5$ ,  $19 // 5$ ,  $19 \% 5$

Each value has a *type*, and only one type, e.g.

- ▶ `int` (integer): `40`, `2`, `-1`, ...
- ▶ `float` (floating-point number): `42.0`, `3.8`, ...
- ▶ `str` (string): `'Hello, world!'`, ...

Find out the type of any value by using the `type()` function:

```
>>> type(84 / 2)  
<class 'float'>
```

# Python

Relational operators produce bool (Boolean) values: True, False.

```
>>> 2 < 3
```

```
True
```

```
>> 2 + 2 == 5
```

```
False
```

```
>>> 'hello' != 'goodbye'
```

```
True
```

Boolean values can be combined with and, or, not.



# Variables, expressions, statements

Like in mathematics, a *variable* is a name that refers to a value.

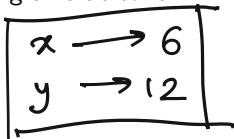
Define a variable by assigning a value to it:

```
>>> x = 6
```

```
>>> x + 1
```

```
7
```

$y = 2 * x$



An *expression* is combination of values, variables, operations that results in ("evaluates to") some value.

A *statement* is a line of code that produces some effect (e.g. assignment).

# Script mode



/home/narain/hello.py

Keyboard shortcuts: Normal ▾

[Share](#)

[Save](#)

[Save as...](#)

[>>> Run](#)



```
1 x = 2*3
2 x + 1
3 print(x/5)
4
```

1.2

>>> 

# Functions

A *function* is a named sequence of statements that performs a computation and may return a value.

e.g. `type(x)` computes the type of the given value and returns it.  
`print(x)` displays the given value as a “side effect”, and does not return anything.

Many useful functions and variables are provided by *modules*, e.g.

```
>>> import math
>>> math.log(2)
0.6931471805599453
>>> 4*math.atan(1) == math.pi
True
```

# Defining functions

We can define our own functions using `def`.

```
def square(x):  
    return x*x
```

The `return` keyword specifies the value output by the function.  
Here, `y = square(5)` sets `y` to 25.

Of course, we can now use this function in other functions:

```
import math  
def hypotenuse(a, b):  
    return math.sqrt(square(a) + square(b))
```



/home/narain/hello.py

Keyboard shortcuts:

Normal

Share

Save

Save as...

>>> Run



```
1 import math
2
3 def square(x):
4     return x*x
5
6 def hypotenuse(a, b):
7     return math.sqrt(square(a) + square(b))
8
9 print(hypotenuse(3, 4))
10
```

```
5.0
>>> hypotenuse(1, 1)
1.4142135623730951
>>> 
```

## Two levels of understanding

1. We know `hypotenuse(a, b)` computes  $\sqrt{\text{square}(a) + \text{square}(b)}$ . We also know `square(x)` computes  $x^2$ , so `hypotenuse(3, 4)` computes  $\sqrt{\text{square}(3) + \text{square}(4)} = \sqrt{3^2 + 4^2} = 5$ .
2. But what actually happens when the expression `hypotenuse(3, 4)` is evaluated?

## Practice exercises

- ▶ Write a function `isTriangle(a, b, c)` to determine if three numbers can form the side lengths of a triangle, using basic operators like `+`, `-`, `<`, `>`, `and`, or.
- ▶ Write a function `solveQuadratic(a, b, c)` to find both roots of a quadratic equation  $ax^2 + bx + c = 0$ .

**Note:** You can return multiple numbers using e.g. `return (x, y)`. A value of the form `(x, y, z, ...)` is called a *tuple* (basically like a Cartesian product).