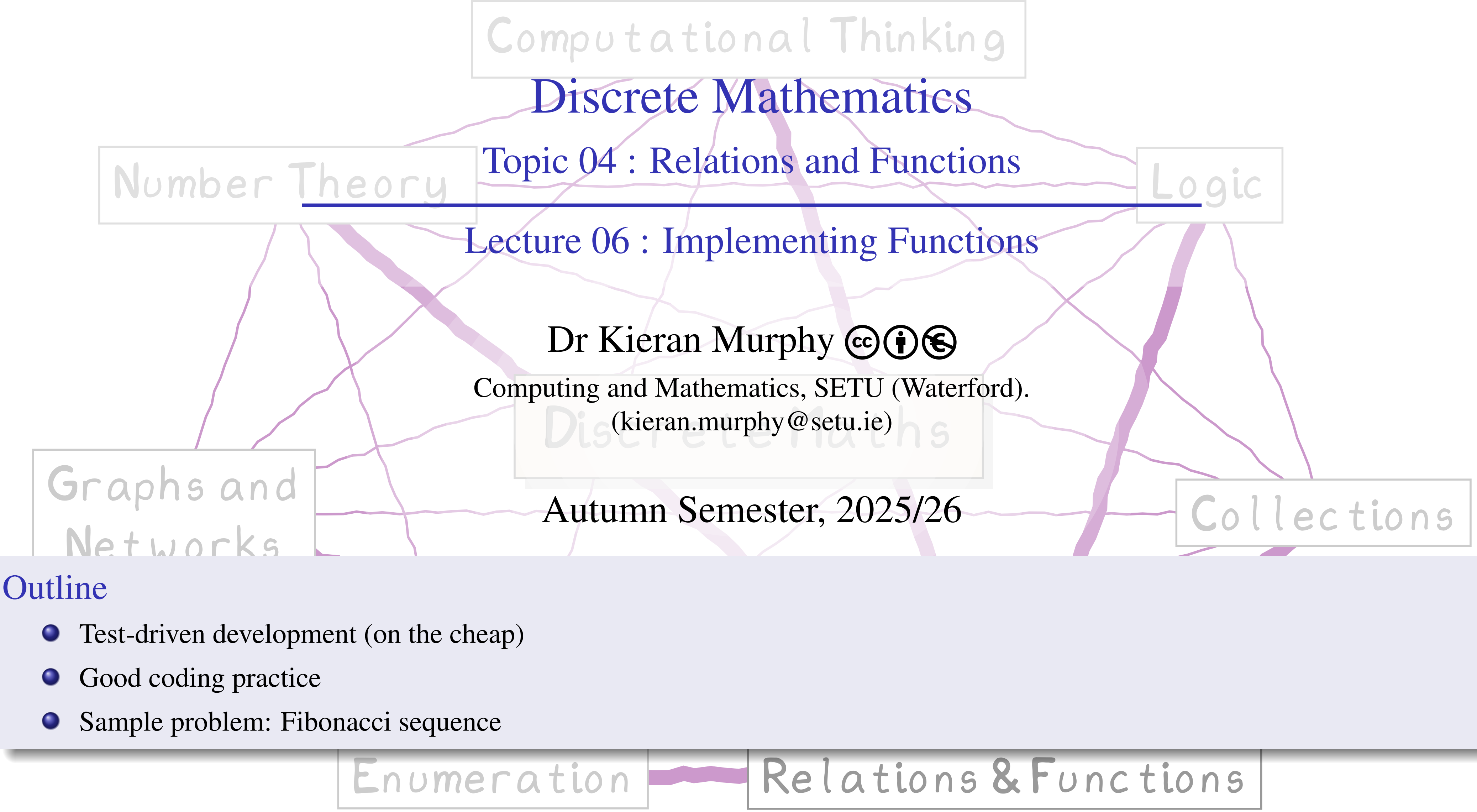


Discrete Mathematics

Topic 04 : Relations and Functions

Lecture 06 : Implementing Functions

Dr Kieran Murphy
Computing and Mathematics, SETU (Waterford).
(kieran.murphy@setu.ie)

Autumn Semester, 2025/26

Outline

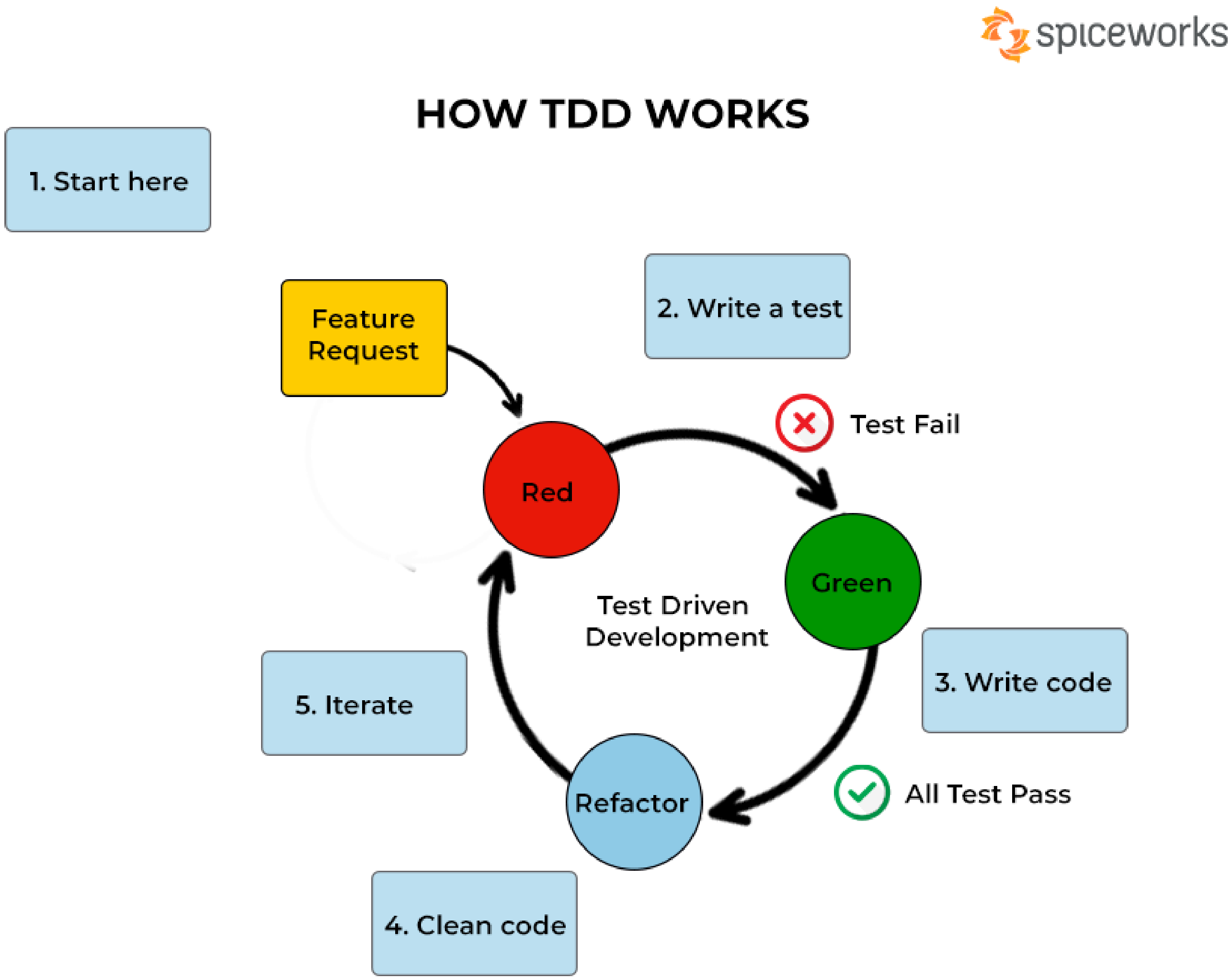
- Test-driven development (on the cheap)
- Good coding practice
- Sample problem: Fibonacci sequence

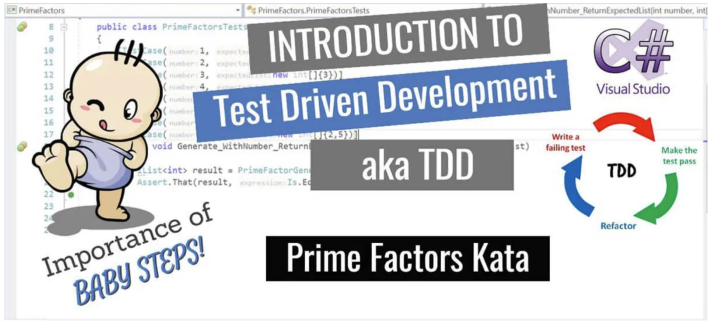
Outline

1. Test Driven Development (TDD) 2

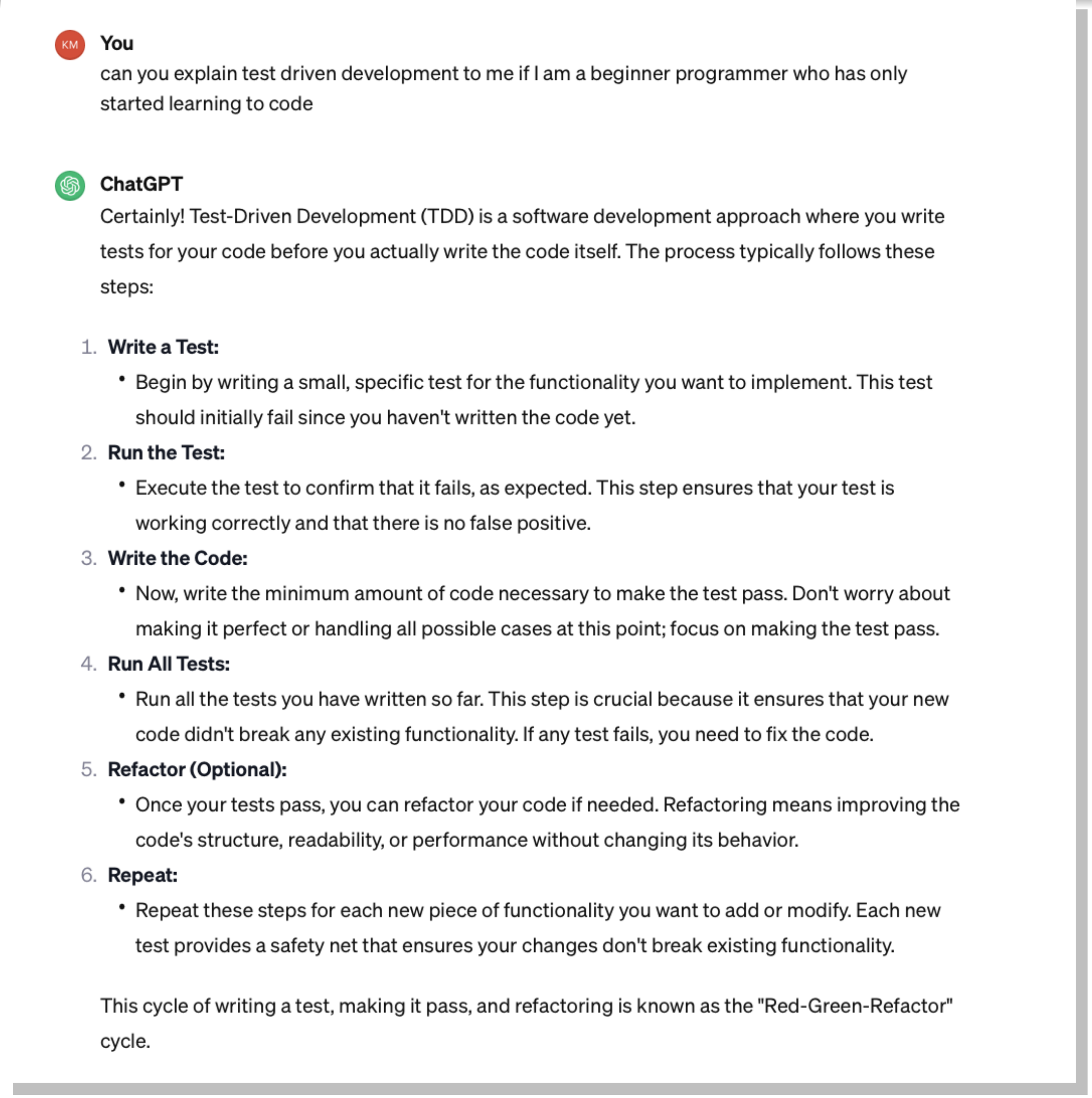
2. Example: Fibonacci Sequence Related Tasks 6

Test Driven Development (TDD) — I have no notes so Google it!





Test Driven Development (TDD) — or ChatGPT it (is this a verb yet?)



You

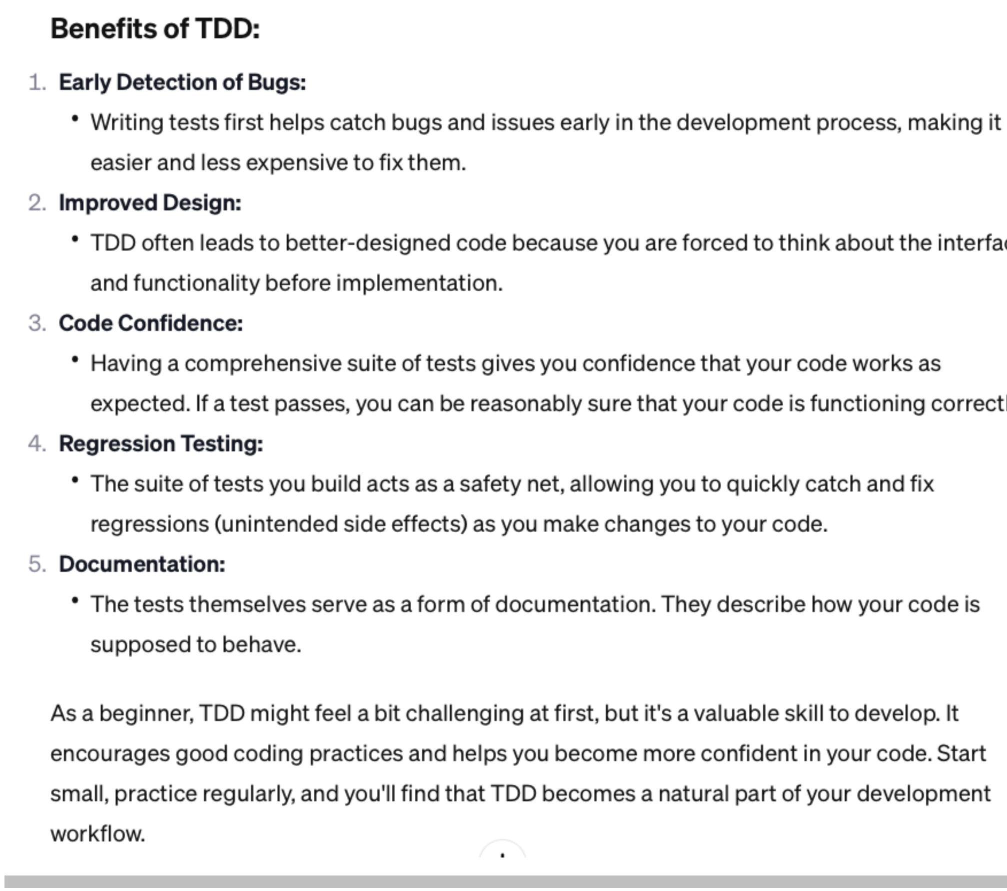
can you explain test driven development to me if I am a beginner programmer who has only started learning to code

ChatGPT

Certainly! Test-Driven Development (TDD) is a software development approach where you write tests for your code before you actually write the code itself. The process typically follows these steps:

1. **Write a Test:**
    - Begin by writing a small, specific test for the functionality you want to implement. This test should initially fail since you haven't written the code yet.
2. **Run the Test:**
    - Execute the test to confirm that it fails, as expected. This step ensures that your test is working correctly and that there is no false positive.
3. **Write the Code:**
    - Now, write the minimum amount of code necessary to make the test pass. Don't worry about making it perfect or handling all possible cases at this point; focus on making the test pass.
4. **Run All Tests:**
    - Run all the tests you have written so far. This step is crucial because it ensures that your new code didn't break any existing functionality. If any test fails, you need to fix the code.
5. **Refactor (Optional):**
    - Once your tests pass, you can refactor your code if needed. Refactoring means improving the code's structure, readability, or performance without changing its behavior.
6. **Repeat:**
    - Repeat these steps for each new piece of functionality you want to add or modify. Each new test provides a safety net that ensures your changes don't break existing functionality.

This cycle of writing a test, making it pass, and refactoring is known as the "Red-Green-Refactor" cycle.



Benefits of TDD:

1. **Early Detection of Bugs:**
    - Writing tests first helps catch bugs and issues early in the development process, making it easier and less expensive to fix them.
2. **Improved Design:**
    - TDD often leads to better-designed code because you are forced to think about the interfa and functionality before implementation.
3. **Code Confidence:**
    - Having a comprehensive suite of tests gives you confidence that your code works as expected. If a test passes, you can be reasonably sure that your code is functioning correct
4. **Regression Testing:**
    - The suite of tests you build acts as a safety net, allowing you to quickly catch and fix regressions (unintended side effects) as you make changes to your code.
5. **Documentation:**
    - The tests themselves serve as a form of documentation. They describe how your code is supposed to behave.

As a beginner, TDD might feel a bit challenging at first, but it's a valuable skill to develop. It encourages good coding practices and helps you become more confident in your code. Start small, practice regularly, and you'll find that TDD becomes a natural part of your development workflow.

Good Coding Practice when Implementing Functions

STEP 1 Verify you understand the task by hand computing the function for multiple input values.

- Pick examples as simply as possible — makes life easier and less error prone.
- These examples will help you understand the task and will help catch implementation errors.

STEP 2 Identify data in (**function parameters**) and data out (**return**).

- This allow us to write the **function signature** and **dummy return value**.

STEP 3 Code any **assert** statements to exclude invalid input.

- An **assert** statement will cause code to crash when its test condition fails.

STEP 4 Wrap the example (inputs,outputs) (found in step 1) in a loop that:

- computes the function using the given input value.
- compares the expected output with the actual output
- outputs appropriate message (success/failure) and test information.

STEP 5 Code function ← finally (see previous slide, i.e., "baby steps")

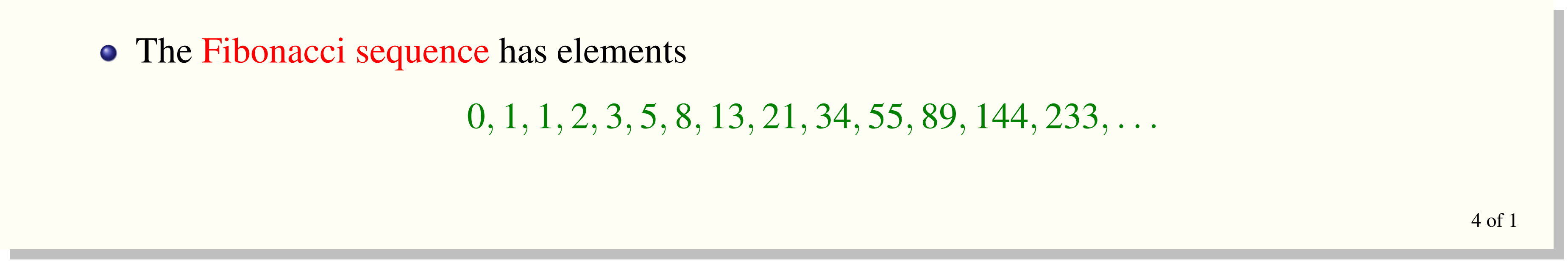
Outline

1. Test Driven Development (TDD) 2

2. Example: Fibonacci Sequence Related Tasks 6

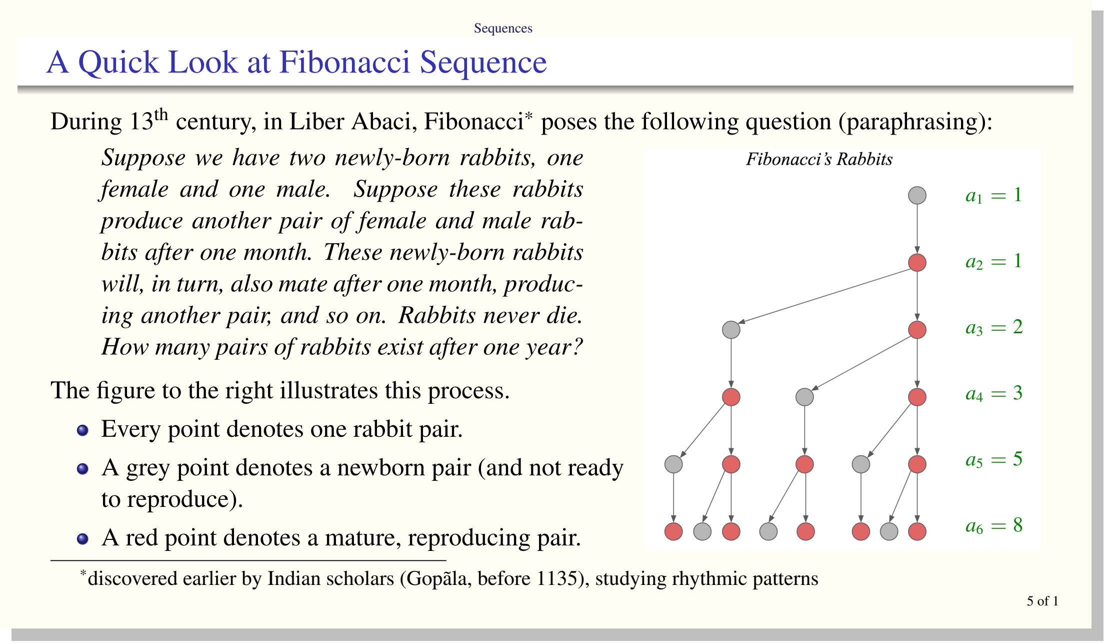
Example: Fibonacci Sequence Related Tasks I

See notes `03-Collections/02-Sequence_Collections`



- The Fibonacci sequence has elements

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, \ldots$$



A Quick Look at Fibonacci Sequence

During 13th century, in Liber Abaci, Fibonacci* poses the following question (paraphrasing):

> *Suppose we have two newly-born rabbits, one female and one male. Suppose these rabbits produce another pair of female and male rabbits after one month. These newly-born rabbits will, in turn, also mate after one month, producing another pair, and so on. Rabbits never die. How many pairs of rabbits exist after one year?*

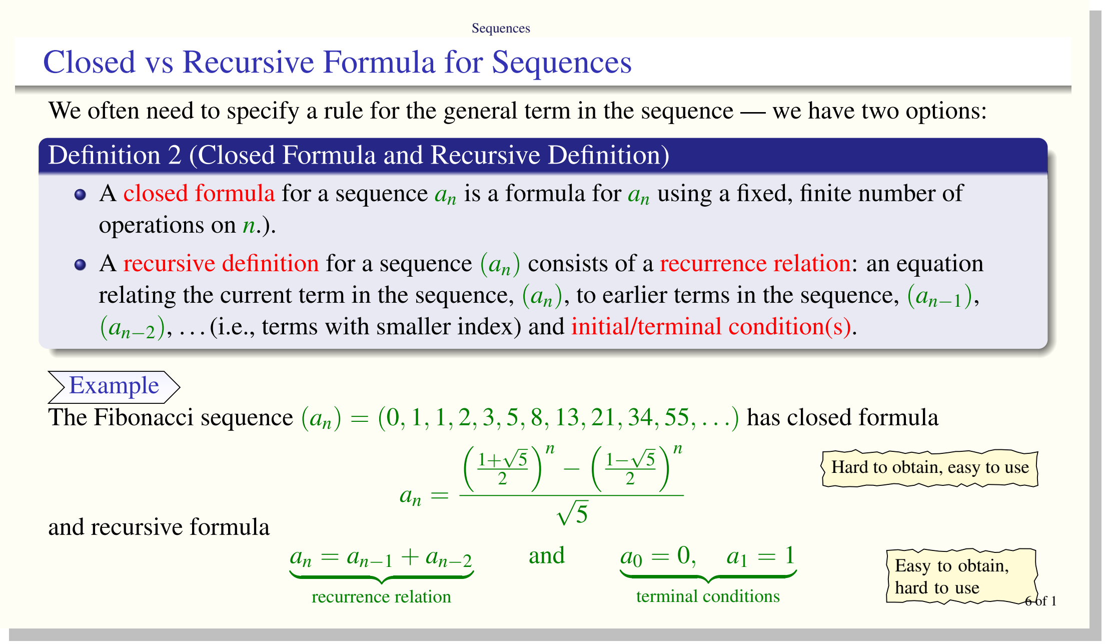
Fibonacci's Rabbits

The figure to the right illustrates this process.

- Every point denotes one rabbit pair.
- A grey point denotes a newborn pair (and not ready to reproduce).
- A red point denotes a mature, reproducing pair.

---

*discovered earlier by Indian scholars (Gopāla, before 1135), studying rhythmic patterns



Closed vs Recursive Formula for Sequences

We often need to specify a rule for the general term in the sequence — we have two options:

Definition 2 (Closed Formula and Recursive Definition)

- A closed formula for a sequence a_n is a formula for a_n using a fixed, finite number of operations on n.).
- A recursive definition for a sequence (a_n) consists of a recurrence relation: an equation relating the current term in the sequence, (a_n), to earlier terms in the sequence, (a_{n-1}), (a_{n-2}), $\ldots$ (i.e., terms with smaller index) and initial/terminal condition(s).

Example

The Fibonacci sequence $(a_n) = (0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots)$ has closed formula

$$a_n = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}}$$

Hard to obtain, easy to use

and recursive formula

$$\underbrace{a_n = a_{n-1} + a_{n-2}}_{\text{recurrence relation}} \quad \text{and} \quad \underbrace{a_0 = 0, \quad a_1 = 1}_{\text{terminal conditions}}$$

Easy to obtain, hard to use

Example: Fibonacci Sequence Related Tasks II

Task 1

Write python function, `fib(n)`, that computes the nth term in the Fibonacci sequence.

Task 2

Write python function, `fibSequence(n1,n2)`, that computes a list containing ${n_1}^{\text{th}}$ (inclusive) up ${n_2}^{\text{th}}$ (exclusive) terms of the Fibonacci sequence.

Task 3

…

Task 1: Compute the nth term in the Fibonacci sequence I

Task 1

Write python function, `fib(n)`, that computes the nth term in the Fibonacci sequence.

STEP 1 Verify you understand the task by hand computing the function for multiple input values.

- Pick examples as simply as possible — makes life easier and less error prone.
- These examples will help you understand the task and will help catch implementation errors.

Some semi-random examples . . .

```
fib(0) = 0
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(10) = 55
⋱
```

so we have test cases

Notice I did not calculate `fib(100)`
Keep examples simple!

n	expected
0	0
1	1
2	1
3	2
10	55

Task 1: Compute the nth term in the Fibonacci sequence II

STEP 2 Identify data in (**function parameters**) and data out (**return**).

- This allow us to write the **function signature** and **dummy return value**.

Function `fib` expects:

- a single parameter, `n`, to be an integer and to be nonnegative (=zero or positive).
- return data is an integer.

STEP 3 Code any **assert** statements to exclude invalid input.

- An **assert** statement will cause code to crash when its test condition fails.

We need to check: (order of tests is important!!!!)

- Parameter `n` is an integer. `type(n)==int`
- Parameter `n` is non-negative. `n>=0`

Task 1: Compute the nth term in the Fibonacci sequence III

STEP 4 Wrap the example (inputs,outputs) (found in step 1) in a loop that:

- computes the function using the given input value.
- compares the expected output with the actual output
- outputs appropriate message (success/failure) and test information.

```python
def fib(n):
    
    assert type(n)==int and n>=0, f"Parameter {n=} should be a nonnegative integer."
    
    # TODO - implement function
    
    return 0


for n,expected in [ (0,0), (1,1), (2,1), (3,2), (10,55)]:
    output = fib(n)
    correct = output==expected
    print(f"{n=}\t {output=}\t {expected=}\t {correct=}")
```

```
n=0  output=0   expected=0 correct=True
n=1  output=0   expected=1 correct=False
n=2  output=0   expected=1 correct=False
n=3  output=0   expected=2 correct=False
n=10     output=0   expected=55    correct=False
```

Task 1: Compute the nth term in the Fibonacci sequence IV

STEP 5 Code function ← finally (see previous slide, i.e., "baby steps")

```python
import math

def fib(n):

    assert type(n)==int and n>=0, f"Parameter {n=} should be a nonnegative integer."

    tmp_1 = (1 + math.sqrt(5)) / 2
    tmp_2 = (1 - math.sqrt(5)) / 2

    return (tmp_1**n - tmp_2**n) / math.sqrt(5)


for n,expected in [ (0,0), (1,1), (2,1), (3,2), (10,55)]:
    output = fib(n)
    correct = output==expected
    print(f"{n=}\t {output=}\t {expected=}\t {correct=}")
```

```
n=0  output=0.0 expected=0 correct=True
n=1  output=1.0 expected=1 correct=True
n=2  output=1.0 expected=1 correct=True
n=3  output=2.0 expected=2 correct=True
n=10     output=55.000000000000014 expected=
correct=False
```

Task 1: Compute the nth term in the Fibonacci sequence V

STEP 5 Code function ← finally (see previous slide, i.e., "baby steps")

```python
import math

def fib(n):

    assert type(n)==int and n>=0, f"Parameter {n=} should be a nonnegative integer."

    tmp_1 = (1 + math.sqrt(5)) / 2
    tmp_2 = (1 - math.sqrt(5)) / 2

    return int( (tmp_1**n - tmp_2**n) / math.sqrt(5) )


for n,expected in [ (0,0), (1,1), (2,1), (3,2), (10,55)]:
    output = fib(n)
    correct = output==expected
    print(f"{n=}\t {output=}\t {expected=}\t {correct=}")
```

```
n=0  output=0   expected=0 correct=True
n=1  output=1   expected=1 correct=True
n=2  output=1   expected=1 correct=True
n=3  output=2   expected=2 correct=True
n=10     output=55  expected=55    correct=Tru
```

Task 2: Compute list of Fibonacci numbers I

Task 2

Write python function, `fibSequence(start,end)`, that computes a list containing *start*th (inclusive) up *end*th (exclusive) terms of the Fibonacci sequence.

STEP 1 Verify you understand the task by hand computing the function for multiple input values.

- Pick examples as simply as possible — makes life easier and less error prone.
- These examples will help you understand the task and will help catch implementation errors.

Some semi-random examples . . .

`fibSequence(0,0) = []`

`fibSequence(0,1) = [0]`

`fibSequence(1,2) = [0,1]`

`fibSequence(6,10) = [8,13,21,34]`

so we have test cases

start	end	expected
0	0	`[]`
0	1	`[0]`
0	2	`[0,1]`
6	10	`[8,13,21,34]`

$\ddots$

Task 2: Compute list of Fibonacci numbers II

STEP 2 Identify data in (**function parameters**) and data out (**return**).

- This allow us to write the **function signature** and **dummy return value**.

Function `fibSequence` expects:

- parameter, `start`, to be a integer and to be nonnegative (=zero or positive).
- parameter, `end`, to be an integer and to be greater or equal to `start`.
- return data is a list.

STEP 3 Code any **assert** statements to exclude invalid input.

- An **assert** statement will cause code to crash when its test condition fails.

We need to check: (order of four tests is important!!!!)

- Parameter `start` is a non-negative integer. `type(start)==int and start>=0`
- Parameter `end` is an integer and not smaller than `start`. `type(end)==int and start<=end`

Task 2: Compute list of Fibonacci numbers III

STEP 4 Wrap the example (inputs,outputs) (found in step 1) in a loop that:

- computes the function using the given input value.
- compares the expected output with the actual output
- outputs appropriate message (success/failure) and test information.

```
def fibSequence(start,end):

    assert type(start)==int and start>=0, f"Parameter {start=} should be a nonnegative integer."
    assert type(end)==int and start<=end, f"Parameter {end=} should be an integer and not smaller

    # TODO - implement function

    return []   # <-- notice empty list


for start,end,expected in [ (0,0,[]), (0,1,[0]), (0,2,[0,1]), (6,10,[8,13,21,34])]:
    output = fibSequence(start,end)
    correct = output==expected
    print(f"{start=}\t {end=}\t {output=}\t {expected=}\t {correct=}")
```

```
start=0 end=0  output=[] expected=[]   correct=True
start=0 end=1  output=[] expected=[0]  correct=False
start=0 end=2  output=[] expected=[0, 1]   correct=False
start=6 end=10 output=[] expected=[8, 13, 21, 34] correct=False
```

Task 2: Compute list of Fibonacci numbers IV

STEP 5 Code function ← finally (see previous slide, i.e., "baby steps")

```
def fibSequence(start,end):

    assert type(start)==int and start>=0, f"Parameter {start=} should be a nonnegative integer."
    assert type(end)==int and start<=end, f"Parameter {end=} should be an integer and not smaller

    return [fib(n) for n in range(start,end)]


for start,end,expected in [ (0,0,[]), (0,1,[0]), (0,2,[0,1]), (6,10,[8,13,21,34])]:
    output = fibSequence(start,end)
    correct = output==expected
    print(f"{start=}\t {end=}\t {output=}\t {expected=}\t {correct=}")
```

```
start=0 end=0  output=[]  expected=[]   correct=True
start=0 end=1  output=[0] expected=[0]  correct=True
start=0 end=2  output=[0, 1] expected=[0, 1]   correct=True
start=6 end=10 output=[8, 13, 21, 34] expected=[8, 13, 21, 34] correct=True
```