Please refer to the online lesson plan on the DT Hub to access all website links and additional resources.

## Learning hook

1. Establish that the first three Fibonacci numbers are 0, 1, 1. One way to do this would be to put an image of a pinecone and the numbers 1, 1, 2, 3, 5, 8 on the board with a question mark and ask students to discuss what they think the lesson is going to look at.
Ensure all students understand the basic Fibonacci rule: Fn = F(n-1) + F(n-2)
2. Have a race where all students are first asked to calculate the 10th Fibonacci number and after that to calculate the 20th Fibonacci number, using a calculator, and stand up when finished. Do not mention the second task until the first is completed! (Students who have not retained prior calculations will be disadvantaged – providing a useful illustration for a method of improving one of our algorithms later in the lesson.)
	* Emphasise that the **first** Fibonacci number is ‘0’.
	* For each round announce student positions as they stand up: first, second, third etc up to around 10th.
3. As you play, ask students to predict how many seconds their computers would take to calculate:
	* the 20th term
	* the 35th term
	* the 100th number.
4. Ask if storing previous calculations assisted with the later one.
5. Ask what an algorithm might look like to perform this task.
	* Ask students what rule could be used to calculate Fibonacci numbers to a given number of terms.
	* Have students use pen and paper to sketch a flowchart to capture their algorithm.

## Learning map and outcomes

Students will compare algorithms used to find the Fibonacci numbers, examine the processes they use and compare their speeds.

Students will determine their favoured algorithm and give reasons for their choice. They will learn to apply this knowledge to new problems.

Students will learn about iterative and recursive algorithms and methods of timing program execution.

You could also focus on the skillset and mindsets that learners mind need to adopt and use during this project, this ties in with the *Creative and Critical Thinking Capabilities.*

## Learning input

1. Explain that we will be comparing algorithms used to find the Fibonacci numbers. We will examine their processes and compare their speeds. We will also learn what a recursive algorithm is, as well as its strengths and limitations. (Below are descriptions of the common approaches that can be used to find Fibonacci numbers. There are many more.)
2. Set students the activity of searching the Internet for algorithms and collecting the names of as many as possible in the time available.
	* Impose the rule that students are not to suggest an algorithm they cannot **explain** to the class (this will limit student choices).
	* Many students will discover webpages summarising many algorithms. This should not be seen as cheating, due to this last stipulation.
	* In this research students will learn about iteration and recursion.

Following this activity the teacher should cover at least the following three methods:

* + Iteration
	+ Recursion
	+ Formula

**Using a Fibonacci iteration algorithm**

1. First, establish with students the basic iterative rule for Fibonacci numbers:

F(n) = F(n-2) + F(n-1)
	* In each case have students check their algorithm works correctly for values of 0, 1, 2 as well as higher values. Discuss the importance of testing boundary conditions and ask students if they exist in this case.
2. Explain that translating the basic Fibonacci rule into an algorithm will necessitate use of a *temp* variable. Lead class reasoning to establish this basic algorithm and also discuss the use of *temp*.

Have students express this in a programming language with which they are familiar and then write it in the form of a*function*. Explain why we use functions here. (We will be developing and comparing a number of Fibonacci algorithms and we want to keep them separated in our program.)

* + Thus in Python:

def Fibonacci\_iteration(n):
a = 0
b = 1
for i in range(0, n):
temp = a
a = b
b = temp + b
return a

1. Revise with the following with students: parameter passing to a function.

Explain: functions run independently of the main program. A running program doesn't see the function, but instead the**value** that the function returns.

To the computer, a variable 'x' doesn't look like an 'x'. It looks instead like the value stored inside ‘x’.

To our program, functions are the same as variables. To the main program they look just like the value they return. A function is like a small program to which parameters are given so it can calculate a result, however our main program only sees the value it returns.
	* Note: Optional aside for more able students. Show students the following side-by-side with earlier iteration. Tell students that this is another Python iteration function:

def fib\_iterative(n):
x,y = 1,0
for k in range(1,n + 1):
x,y = y,x+y
return y

Most students will assume this is equivalent to the earlier Python iteration function above. This is not the case. Python always evaluates the right-hand side first.

* + Further:

If x=2, y=3
and if
x,y = y,x+y
then the result is
x=3
y=5

* + Whereas:

if x=2, y=3
x=y
y=x+y
then the result is
x=3>
y=6

* + The first algorithm is actually equivalent to the example following, using a *temp* to hold the x value:

if x=2, y=3
temp = x
x = y
y= temp + x
then the result is
x=3
y=5

1. It will be valuable to give the above as a class exercise.

**Using a Fibonacci recursion algorithm

A joke popular among programmers is that in order to understand recursion, you must first understand recursion! It is also said that people go through three phases while learning recursion:**

* First, they hate it, because they can't understand it.
* Then, they love it because they finally understand it.
* Finally, they hate it again because they decide it's inefficient!

1. Give students the following example of recursion:

	* First explain what a factorial is. Tell the students:
	What if we are asked to find 5!
	We know this is equal to 5.4!
	Now we need to find 4! Which is 4.3!
	Now we need to find 3! Which is 3.2!
	Now we need to find 2! Which is 2.1! or 2x1=2
	But we know 2! = 2
	So working back up we can say 3!= 3x2=6
	And up again 4!=4x6=24
	And up again 5!= 5x24=120
	Answer: Thus final result is 120
2. Here is a recursive Python function for Fibonacci:

def fib (n):
if n < 2: return n
return fib(n - 1)+ fib(n - 2)

1. Have students try to work through this on paper using n = 6 and then illustrate with the aid of a tree diagram as follows.
	* To perform recursion in order to solve FIB(4), we are forced to descend the tree to cases below the one we want to find, until we eventually reach those shown shaded, whose values we know, and so, ‘carrying’ these we ascend the tree using addition to resolve each higher FIB(n) value, until we eventually reach the top of the tree with our solution:
	
	* When tracing most recursive functions, there is a descending and ascending part.
	* The descending part occurs as the recursion needs to ask questions for which the only answers are lower down, so it heads to the base cases (which are known). The ascending part occurs when the recursion returns to the original again carrying the answers.
	* Suppose we have a function **a** which we wish to evaluate, which must make a call to function **b** to find its own value, but function **b** must make a call to function **c**, and function **c** must make a call to function **d**. Now let us say the value of **d** is known. Once function **d** is done, it delivers the answer back to **c**, which delivers it to **b**, which finally delivers it back to **a**.
	In going from **a** to **d** it descends with the burden of a question and then it ascends carrying an answer.
	* The same thing happens with recursive functions: function **f** makes a recursive call to itself - function **f**, which makes a call to function **f**, which makes a call to function **f** (the base case), then it eventually ascends back to the original **f**.
	Note: Students will discover **for themselves in the Learning** construction phase that recursion is a very, very slow method for larger terms in Fibonacci. It is slow because it is inefficient. In fact, for terms greater than 40 it is virtually unusable. However, students should not dismiss recursion as it is a superior **choice for other algorithms, such as some sorting algorithms.**
2. Recursion forces a program to calculate the same values over and over again. Using a feature of Python called a*dictionary* we can store previously calculated values and save a large amount of time:

fibTable = {0:0, 1:1}
def fib\_dict(n):
if n in fibTable:
return fibTable[n]
else:
fibTable[n] = fib\_dict(n-1) + fib\_dict(n-2)
return fibTable[n]

* + This is demonstrated in our three sample programs.

**Using the Binet formula**

Explain to students that this formula must round off values as it uses floating point numbers:

\[F(n) = (\frac{1+\sqrt{5})^n-(1-\sqrt{5})^n}{2^n\sqrt{5}}\]

This formula was known to others but was derived in 1843 by Jacques Binet.

## Learning construction

1. Explain to students that they are to construct a program which calls for at least three different *algorithms* or *methods* of calculating any given term of the Fibonacci sequence of integers. Explain that we will all agree that the first three terms are to be 0, 1, 1 and that the positions of these are 0, 1 and 2 respectively. Marking the first term as ‘0’ will prove helpful later.
2. Students design and code their programs using their three chosen algorithms.
3. Students thoroughly test their programs.
4. Students place a timer in their program to evaluate the processing time for each method.
	* It is suggested that students set:

start\_time = time.clock()

before performing each particular Fibonacci algorithm, then set:

end\_time = time.clock()

after it ends.

* + Thus elapsed time will be:

elapsed\_time = end\_time - start\_time

We will discover later that this is a rather inexact method of timing.

1. Discuss with students where such a calculation should be placed in their program.
	* They should allow users to state which term is to be evaluated.
	* They are to allow the user to repeat the program without quitting and restarting it.
	* Their goal is to discover which algorithms are most efficient in terms of time.
2. Make sure students properly compare times, which will likely be expressed in varying powers of 10. It will be necessary for students to re-express values in the same power of 10.
3. Have groups of students compare results using similar algorithms on different computers.
	* Students use a table to record their findings detailing which algorithms are most efficient for selected terms in particular ranges of Fibonacci terms. Suggested terms to test in particular ranges:
	0-5, test 0, 5
	6–10, test 10
	11–15, test 15
	16–20, test 20
	21–25, test 25
	26–30, test 30
	31–35, test 35
	36–40, test 40
	41–45, test 45
	46–100, test 100
	>100, test 200, 500, 1000
	* Ask students to determine their favoured algorithms and give reasons for their choices. A spreadsheet is recommended to keep track of the times. See example program: Fibonacci\_7\_ways\_1.py

**Improving the time**

Students will discover that the times they get will vary each time they run their program.

1. Ask them what factors might affect the time measurement (CPU activity, other running programs, PRINT statements, etc).
	* Ask students to type in some miscellaneous PRINT statements into the middle of one of their functions and note its effect on the time taken.
	* Ask how this problem can be addressed. Can they think of a way of ‘evening out’ the interruptions the CPU may experience? (Perform each function say, 10 times, and then average the total of all 10 times, or even more times.)
	See example program: *Fibonacci 2 (PY)*
	Time taken will depend upon whether the term is small or large. Recursion algorithms for Fibonacci are notoriously inefficient. Students should also observe that the times are much shorter when the timer is started in the function itself.
	As in example program: *Fibonacci 2 (PY)*
	and not in the main program as is the case in example program: *Fibonacci 1 (PY)*
2. Students report their findings.

 **Improving the timer further**

Most students will have added identical timing code into each of their functions.

1. Discuss how repeated code such as this would be best written as a separate function to be called when needed (efficiency; ease of reading the code; avoidance of errors if changes are made – such as increasing the number of repeats for the calculations).

	* The sample program provided: *Fibonacci 3 (PY)*
	shows an example of the timer as a separate function called by each Fibonacci algorithm.

## Learning demo

Demonstrate the Python programs supplied:
Fibonacci 1 (PY); Fibonacci 2 (PY); Fibonacci 3 (PY)
Students compare and contrast these with their own.

## Curriculum links

| Links with Digital Technologies Curriculum Area |
| --- |
| **Strand** | **Content Description** |
| **Knowledge and Understanding** | Implement, modify and debug modular programs, applying selected algorithms and data structures, including in an object-oriented programming language (AC9TDI10P09). |

## Assessment

|  | **Quantity of Knowledge** | **Quality of Understanding** |
| --- | --- | --- |
| **Image compression** | No evidence of understanding | Student is able to build a working program containing at least one Fibonacci algorithms | Student is able to build a working program containing three Fibonacci algorithms | Student is able to build a working program, efficiently timing three Fibonacci algorithms | Student is able to compare the efficiency of at least three algorithms for a range of high and low values, for a variety of functions, one of which uses recursion. |
| **Optional Score** | 0 | 1 | 2 | 3 | 4 |