# [Introduction to Data Structures and Algorithms](https://www.w3schools.com/dsa/index.php)

****Data Structures**** is about how data can be stored in different structures.

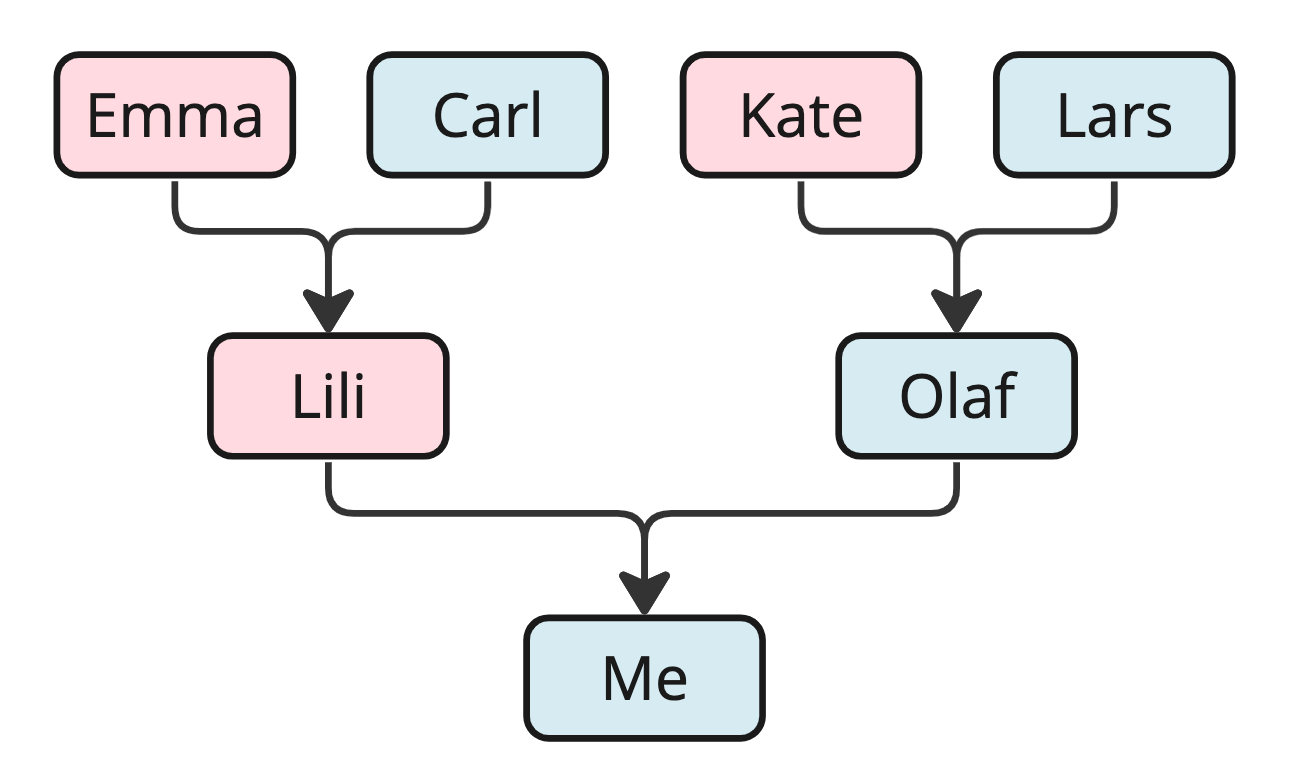
****Algorithms**** is about how to solve different problems, often by searching through and manipulating data structures.

Theory about Data Structures and Algorithms (DSA) helps us to use large amounts of data to solve problems efficiently.

## What are Data Structures?

A data structure is a way to store data.

We structure data in different ways depending on what data we have, and what we want to do with it.



First, let's consider an example without computers in mind, just to get the idea.

If we want to store data about people we are related to, we use a family tree as the data structure. We choose a family tree as the data structure because we have information about people we are related to and how they are related, and we want an overview so that we can easily find a specific family member, several generations back.

With such a family tree data structure visually in front of you, it is easy to see, for example, who my mother's mother is—it is 'Emma,' right? But without the links from child to parents that this data structure provides, it would be difficult to determine how the individuals are related.

Data structures give us the possibility to manage large amounts of data efficiently for uses such as large databases and internet indexing services.

Data structures are essential ingredients in creating fast and powerful algorithms. They help in managing and organizing data, reduce complexity, and increase efficiency.

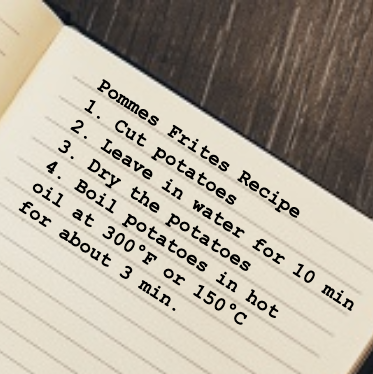
In Computer Science there are two different kinds of data structures.

****Primitive Data Structures**** are basic data structures provided by programming languages to represent single values, such as integers, floating-point numbers, characters, and booleans.

****Abstract Data Structures**** are higher-level data structures that are built using primitive data types and provide more complex and specialized operations. Some common examples of abstract data structures include arrays, linked lists, stacks, queues, trees, and graphs.

## What are Algorithms?

An algorithm is a set of step-by-step instructions to solve a given problem or achieve a specific goal.



A cooking recipe written on a piece of paper is an example of an algorithm, where the goal is to make a certain dinner. The steps needed to make a specific dinner are described exactly.

When we talk about algorithms in Computer Science, the step-by-step instructions are written in a programming language, and instead of food ingredients, an algorithm uses data structures.

Algorithms are fundamental to computer programming as they provide step-by-step instructions for executing tasks. An efficient algorithm can help us to find the solution we are looking for, and to transform a slow program into a faster one.

By studying algorithms, developers can write better programs.

Algorithm examples:

* Finding the fastest route in a GPS navigation system
* Navigating an airplane or a car (cruise control)
* Finding what users search for (search engine)
* Sorting, for example sorting movies by rating

The algorithms we will look at in this tutorial are designed to solve specific problems, and are often made to work on specific data structures. For example, the 'Bubble Sort' algorithm is designed to sort values, and is made to work on arrays.

## Data Structures together with Algorithms

Data structures and algorithms (DSA) go hand in hand. A data structure is not worth much if you cannot search through it or manipulate it efficiently using algorithms, and the algorithms in this tutorial are not worth much without a data structure to work on.

DSA is about finding efficient ways to store and retrieve data, to perform operations on data, and to solve specific problems.

By understanding DSA, you can:

* Decide which data structure or algorithm is best for a given situation.
* Make programs that run faster or use less memory.
* Understand how to approach complex problems and solve them in a systematic way.

## Where is Data Structures and Algorithms Needed?

Data Structures and Algorithms (DSA) are used in virtually every software system, from operating systems to web applications:

* For managing large amounts of data, such as in a social network or a search engine.
* For scheduling tasks, to decide which task a computer should do first.
* For planning routes, like in a GPS system to find the shortest path from A to B.
* For optimizing processes, such as arranging tasks so they can be completed as quickly as possible.
* For solving complex problems: From finding the best way to pack a truck to making a computer 'learn' from data.

DSA is fundamental in nearly every part of the software world:

* Operating Systems
* Database Systems
* Web Applications
* Machine Learning
* Video Games
* Cryptographic Systems
* Data Analysis
* Search Engines

## Theory and Terminology

As we go along in this tutorial, new theoretical concepts and terminology (new words) will be needed so that we can better understand the data structures and algorithms we will be working on.

These new words and concepts will be introduced and explained properly when they are needed, but here is a list of some key terms, just to get an overview of what is coming:

|  |  |
| --- | --- |
| Term | Description |
| Algorithm | A set of step-by-step instructions to solve a specific problem. |
| Data Structure | A way of organizing data so it can be used efficiently. Common data structures include arrays, linked lists, and binary trees. |
| Time Complexity | A measure of the amount of time an algorithm takes to run, depending on the amount of data the algorithm is working on. |
| Space Complexity | A measure of the amount of memory an algorithm uses, depending on the amount of data the algorithm is working on. |
| Big O Notation | A mathematical notation that describes the limiting behavior of a function when the argument tends towards a particular value or infinity. Used in this tutorial to describe the time complexity of an algorithm. |
| Recursion | A programming technique where a function calls itself. |
| Divide and Conquer | A method of solving complex problems by breaking them into smaller, more manageable sub-problems, solving the sub-problems, and combining the solutions. Recursion is often used when using this method in an algorithm. |
| Brute Force | A simple and straight forward way an algorithm can work by simply trying all possible solutions and then choosing the best one. |

## Where to Start?

In this tutorial, you will first learn about a data structure with matching algorithms, before moving on to the next data structure.

Further into the tutorial the concepts become more complex, and it is therefore a good idea to learn DSA by doing the tutorial step-by-step from the start.

And as mentioned on the previous page, you should be comfortable in at least one of the most common programming languages, like for example [JavaScript](https://www.w3schools.com/js/default.asp), [C](https://www.w3schools.com/c/default.asp) or [Python](https://www.w3schools.com/python/default.asp), before doing this tutorial.

On the next page we will look at two different algorithms that prints out the first 100 Fibonacci numbers using only primitive data structures (two integer variables). One algorithm uses a loop, and one algorithm uses something called recursion

# A Simple Algorithm

[❮ Previous](https://www.w3schools.com/dsa/dsa_intro.php)[Next ❯](https://www.w3schools.com/dsa/dsa_data_arrays.php)

## Fibonacci Numbers

The Fibonacci numbers are very useful for introducing algorithms, so before we continue, here is a short introduction to Fibonacci numbers.

The Fibonacci numbers are named after a 13th century Italian mathematician known as Fibonacci.

The two first Fibonacci numbers are 0 and 1, and the next Fibonacci number is always the sum of the two previous numbers, so we get 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

Create fibonacci numbers.

 Done!

0

1

1

2

3

5

8

13

21

34

55

89

This tutorial will use loops and recursion a lot. So before we continue, let's implement three different versions of the algorithm to create Fibonacci numbers, just to see the difference between programming with loops and programming with recursion in a simple way.

## The Fibonacci Number Algorithm

To generate a Fibonacci number, all we need to do is to add the two previous Fibonacci numbers.

The Fibonacci numbers is a good way of demonstrating what an algorithm is. We know the principle of how to find the next number, so we can write an algorithm to create as many Fibonacci numbers as possible.

Below is the algorithm to create the 20 first Fibonacci numbers.

****How it works:****

1. Start with the two first Fibonacci numbers 0 and 1.
   1. Add the two previous numbers together to create a new Fibonacci number.
   2. Update the value of the two previous numbers.
2. Do point a and b above 18 times.

## Loops vs Recursion

To show the difference between loops and recursion, we will implement solutions to find Fibonacci numbers in three different ways:

1. An implementation of the Fibonacci algorithm above using a for loop.
2. An implementation of the Fibonacci algorithm above using recursion.
3. Finding the *[Math Processing Error]*th Fibonacci number using recursion.

## 1. Implementation Using a For Loop

It can be a good idea to list what the code must contain or do before programming it:

* Two variables to hold the previous two Fibonacci numbers
* A for loop that runs 18 times
* Create new Fibonacci numbers by adding the two previous ones
* Print the new Fibonacci number
* Update the variables that hold the previous two fibonacci numbers

Using the list above, it is easier to write the program:

### Example

prev2 = 0

prev1 = 1

print(prev2)

print(prev1)

for fibo in range(18):

newFibo = prev1 + prev2

print(newFibo)

prev2 = prev1

prev1 = newFibo

[Run Example »](https://www.w3schools.com/dsa/trydsa.php?filename=demo_findfibo1)

## 2. Implementation Using Recursion

Recursion is when a function calls itself.

To implement the Fibonacci algorithm we need most of the same things as in the code example above, but we need to replace the for loop with recursion.

To replace the for loop with recursion, we need to encapsulate much of the code in a function, and we need the function to call itself to create a new Fibonacci number as long as the produced number of Fibonacci numbers is below, or equal to, 19.

Our code looks like this:

### Example

print(0)

print(1)

count = 2

def fibonacci(prev1, prev2):

global count

if count <= 19:

newFibo = prev1 + prev2

print(newFibo)

prev2 = prev1

prev1 = newFibo

count += 1

fibonacci(prev1, prev2)

else:

return

fibonacci(1,0)

[Run Example »](https://www.w3schools.com/dsa/trydsa.php?filename=demo_findfibo2)

## 3. Finding The *[Math Processing Error]*th Fibonacci Number Using Recursion

To find the *[Math Processing Error]*th Fibonacci number we can write code based on the mathematic formula for Fibonacci number *[Math Processing Error]*:

*[Math Processing Error]*

This just means that for example the 10th Fibonacci number is the sum of the 9th and 8th Fibonacci numbers.

****Note:****This formula uses a 0-based index. This means that to generate the 20th Fibonacci number, we must write *[Math Processing Error]*.

When using this concept with recursion, we can let the function call itself as long as *[Math Processing Error]* is less than, or equal to, 1. If *[Math Processing Error]* it means that the code execution has reached one of the first two Fibonacci numbers 1 or 0.

The code looks like this:

### Example

def F(n):

if n <= 1:

return n

else:

return F(n - 1) + F(n - 2)

print(F(19))

[Run Example »](https://www.w3schools.com/dsa/trydsa.php?filename=demo_findfibo3)

Notice that this recursive method calls itself two times, not just one. This makes a huge difference in how the program will actually run on our computer. The number of calculations will explode when we increase the number of the Fibonacci number we want. To be more precise, the number of function calls will double every time we increase the Fibonacci number we want by one.

Just take a look at the number of function calls for *[Math Processing Error]*:

To better understand the code, here is how the recursive function calls return values so that *[Math Processing Error]* returns the correct value in the end:

There are two important things to notice here: The amount of function calls, and the amount of times the function is called with the same arguments.

So even though the code is fascinating and shows how recursion work, the actual code execution is too slow and ineffective to use for creating large Fibonacci numbers.

## Summary

Before we continue, let's look at what we have seen so far:

* An algorithm can be implemented in different ways and in different programming languages.
* Recursion and loops are two different programming techniques that can be used to implement algorithms.

It is time to move on to the first data structure we will look at, the array.

Click the "Next" button to continue.

## DSA Exercises

## Test Yourself With Exercises

## Exercise:

How can we make this fibonacci() function recursive?

print(0)

print(1)

count = 2

def fibonacci(prev1, prev2):

global count

if count <= 19:

newFibo = prev1 + prev2

print(newFibo)

prev2 = prev1

prev1 = newFibo

count += 1

(prev1, prev2)

else:

return

fibonacci(1,0)