

# DSA Tutorial

[Home](#)[Next](#)

## Learn Data Structures and Algorithms

Data Structures and Algorithms (DSA) is a fundamental part of Computer Science that teaches you how to think and solve complex problems systematically.

Using the right data structure and algorithm makes your program run faster, especially when working with lots of data.

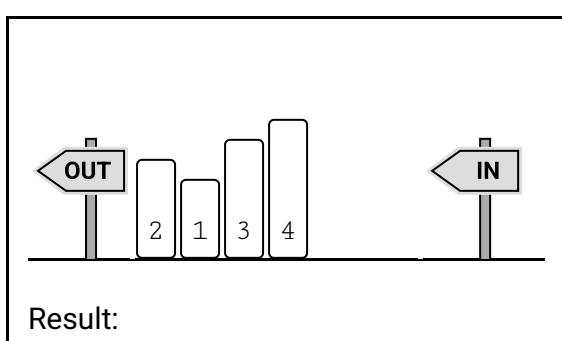
Knowing DSA can help you perform better in job interviews and land great jobs in tech companies.

[Start learning DSA now »](#)

## This Tutorial

This tutorial is made to help you learn Data Structures and Algorithms (DSA) fast and easy.

Animations, like the one below, are used to explain ideas along the way.



First, you will learn the fundamentals of DSA: understanding different data structures, basic algorithm concepts, and how they are used in programming.

Then, you will learn more about complex data structures like trees and graphs, study advanced sorting and searching algorithms, explore concepts like time complexity, and more.

This tutorial will give you a solid foundation in Data Structures and Algorithms, an essential skill for any software developer.

---

## Try it Yourself Examples in Every Chapter

In every chapter, you can edit the examples online, and click on a button to view the result.

The code examples in this tutorial are written in Python, C, and Java. You can see this by clicking the "Run Example" button.

### Example

```
my_array = [7, 12, 9, 4, 11]
minVal = my_array[0]

for i in my_array:
    if i < minVal:
        minVal = i

print('Lowest value:',minVal)
```

[Run Example »](#)

[Tutorials ▾](#)[Exercises ▾](#)[Services ▾](#)[Sign Up](#)[Log in](#)[☰](#) [TLIN](#) [SASS](#) [VUE](#) [DSA](#) [GEN AI](#) [SCIPY](#) [AWS](#) [CYBERSECURITY](#) [DATA SC](#)

# What You Should Already Know

Although Data Structures and Algorithms is actually not specific to any programming language, you should have a basic understanding of programming in one of these common programming languages:

- [Python](#)
- [C](#)
- [C++](#)
- [Java](#)
- [JavaScript](#)

## DSA History

The word 'algorithm' comes from 'al-Khwarizmi', named after a Persian scholar who lived around year 800.

The concept of algorithmic problem-solving can be traced back to ancient times, long before the invention of computers.

The study of Data Structures and Algorithms really took off with the invention of computers in the 1940s, to efficiently manage and process data.

Today, DSA is a key part of Computer Science education and professional programming, helping us to create faster and more powerful software.

## DSA Exercises

# Introduction to Data Structures and Algorithms

[« Previous](#)[Next »](#)

**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.

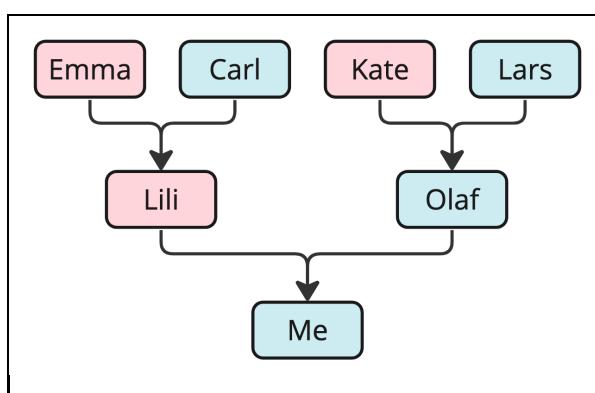
## What are Data Structures?

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



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:

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.

- 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.

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](#), [C](#) or [Python](#), 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.

Click the 'Next' button to continue.

[!\[\]\(220899daa07e43db9eb76860c91c848f\_img.jpg\) Previous](#)[Next !\[\]\(ade431365d5a245a24c736a8cc4219e3\_img.jpg\)](#)

**W3schools Pathfinder**

Track your progress - it's free!

[Sign Up](#) [Log in](#)



# A Simple Algorithm

[« Previous](#)[Next »](#)

## 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.

[Create](#)

0  1

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

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.
  - a. Add the two previous numbers together to create a new Fibonacci number.
  - b. 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  $n$ 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



VUE DSA

GEN AI

SCIPY

AWS

CYBERSECURITY

DATA SCIENCE

```
print(prev2)
print(prev1)
for fibo in range(18):
    newFibo = prev1 + prev2
    print(newFibo)
    prev2 = prev1
    prev1 = newFibo
```

[Run Example »](#)

## 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(1, 0)

[Run Example »](#)

## 3. Finding The nth Fibonacci Number Using Recursion

To find the nth Fibonacci number we can write code based on the mathematic formula for Fibonacci number n:

$$F(n) = F(n - 1) + F(n - 2)$$

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 F(19).

When using this concept with recursion, we can let the function call itself as long as n is less than, or equal to, 1. If  $n \leq 1$  it means that the code execution has reached one of the first two Fibonacci numbers 1 or 0.

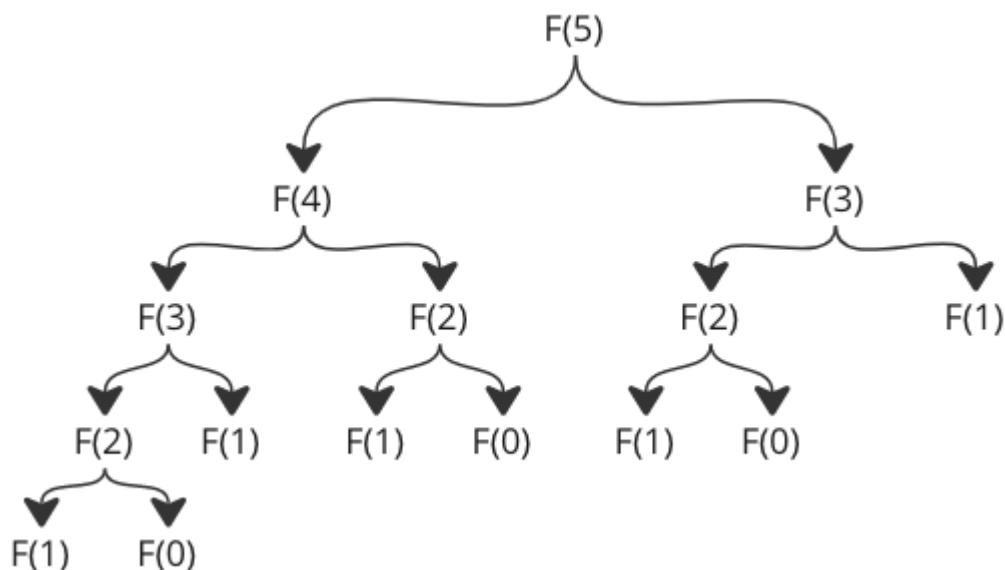
The code looks like this:

### Example

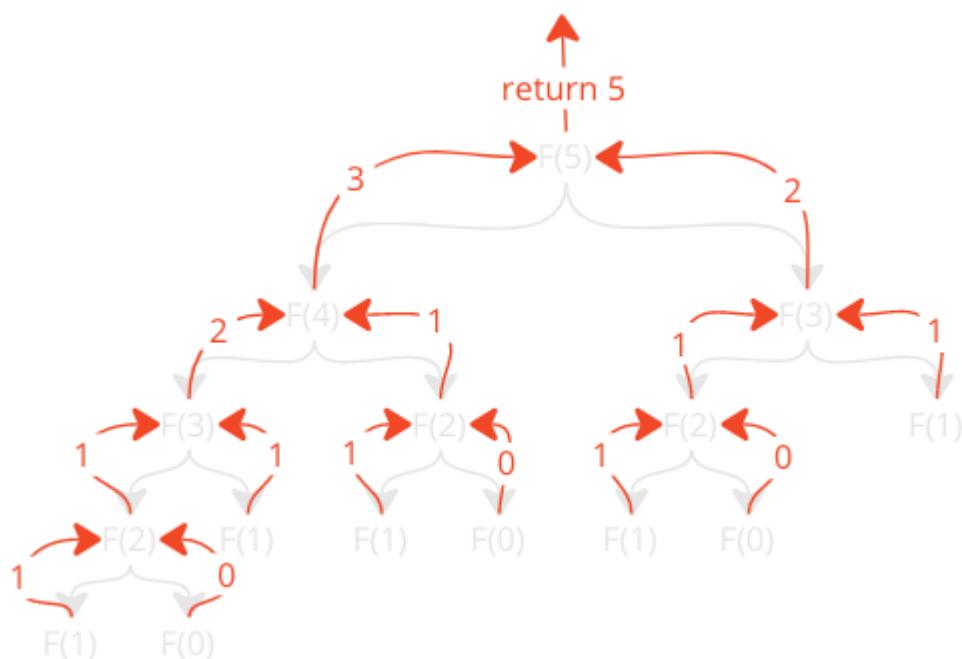
```
def F(n):
    if n <= 1:
        return n
    else:
        return F(n - 1) + F(n - 2)
```

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  $F(5)$ :



To better understand the code, here is how the recursive function calls return values so that  $F(5)$  returns the correct value in the end:



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)
```

# DSA Arrays

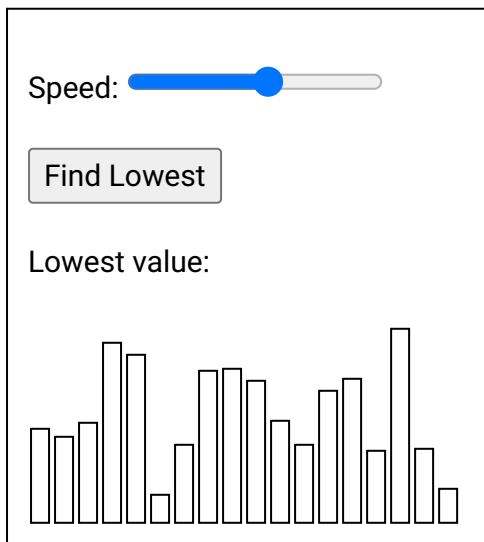
[« Previous](#)[Next »](#)

## Arrays

An array is a data structure used to store multiple elements.

Arrays are used by many algorithms.

For example, an algorithm can be used to look through an array to find the lowest value, like the animation below shows:



In Python, an array can be created like this:

**Note:** The Python code above actually generates a Python 'list' data type, but for the scope of this tutorial the 'list' data type can be used in the same way as an array. Learn more about Python lists [here](#).

Arrays are indexed, meaning that each element in the array has an index, a number that says where in the array the element is located. The programming languages in this tutorial (Python, Java, and C) use zero-based indexing for arrays, meaning that the first element in an array can be accessed at index 0.

In Python, this code use index 0 to write the first array element (value 7) to the console:

## Example

Python:

```
my_array = [7, 12, 9, 4, 11]
print( my_array[0] )
```

[Run Example »](#)

## Algorithm: Find The Lowest Value in an Array

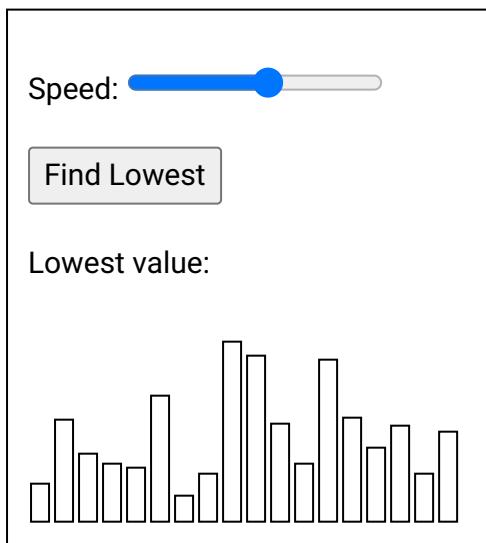
Let's create our first algorithm using the array data structure.

Below is the algorithm to find the lowest number in an array.

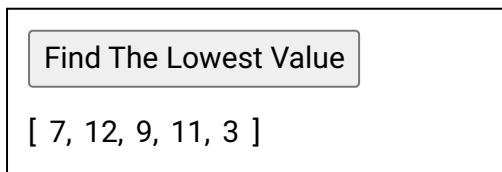
### How it works:

1. Go through the values in the array one by one.
2. Check if the current value is the lowest so far, and if it is, store it.

Try the simulation below to see how the algorithm for finding the lowest value works (the animation is the same as the one on the top of this page):



This next simulation also finds the lowest value in an array, just like the simulation above, but here we can see how the numbers inside the array are checked to find the lowest value:



## Implementation

Before implementing the algorithm using an actual programming language, it is usually smart to first write the algorithm as a step-by-step procedure.

If you can write down the algorithm in something between human language and programming language, the algorithm will be easier to implement later because we avoid drowning in all the details of the programming language syntax.

1. Create a variable 'minVal' and set it equal to the first value of the array.
2. Go through every element in the array.
3. If the current element has a lower value than 'minVal', update 'minVal' to this value.

You can also write the algorithm in a way that looks more like a programming language if you want to, like this:

```
Variable 'minVal' = array[0]
For each element in the array
    If current element < minVal
        minVal = current element
```

**Note:** The two step-by-step descriptions of the algorithm we have written above can be called 'pseudocode'. Pseudocode is a description of what a program does, using language that is something between human language and a programming language.

After we have written down the algorithm, it is much easier to implement the algorithm in a specific programming language:

## Example

Python:

```
my_array = [7, 12, 9, 4, 11]
minVal = my_array[0]      # Step 1

for i in my_array:        # Step 2
    if i < minVal:        # Step 3
        minVal = i

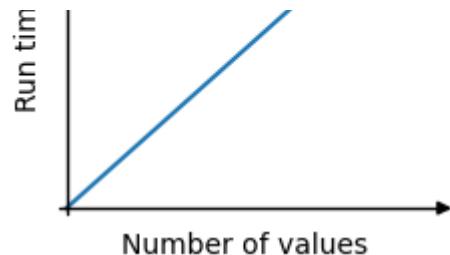
print('Lowest value: ',minVal) # Step 4
```

[Run Example »](#)

## Algorithm Time Complexity

☰ TLIN SASS VUE DSA GEN AI SCIPY AWS CYBERSECURITY DATA SC

In the example above, the time the algorithm needs to run is proportional, or linear, to the size of the data set. This is because the algorithm must visit every array element one time to find the lowest value. The loop must run 5 times since there are 5 values in the array. And if the array had 1000 values, the loop would have to run 1000 times.



Try the simulation below to see this relationship between the number of compare operations needed to find the lowest value, and the size of the array.

See [this page](#) for a more thorough explanation of what time complexity is.

Each algorithm in this tutorial will be presented together with its time complexity.

Set values:  300

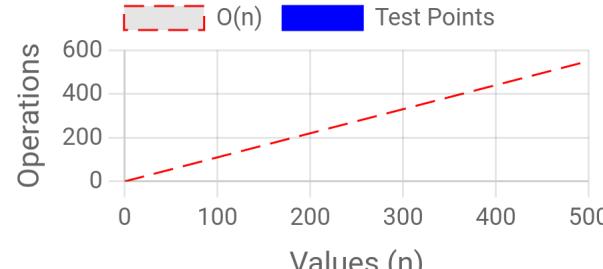
Random  
 Descending  
 Ascending  
 10 Random

Operations: 0

Run Clear

Operations (Y-axis) vs Values (n) (X-axis)

Legend:    $O(n)$    Test Points



Values (n)	Operations
0	0
100	~100
200	~200
300	~300
400	~400
500	~500

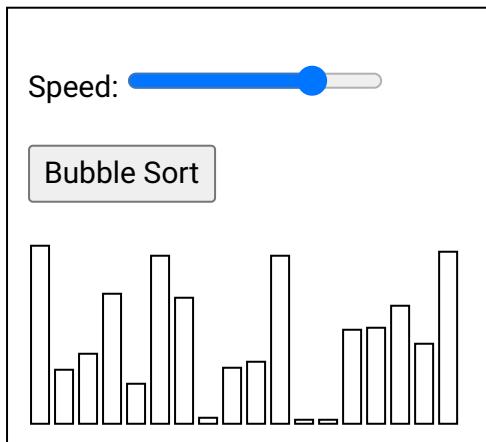
## DSA Exercises

# DSA Bubble Sort

[« Previous](#)[Next »](#)

## Bubble Sort

Bubble Sort is an algorithm that sorts an array from the lowest value to the highest value.



Run the simulation to see how it looks like when the Bubble Sort algorithm sorts an array of values. Each value in the array is represented by a column.

The word 'Bubble' comes from how this algorithm works, it makes the highest values 'bubble up'.

### How it works:

1. Go through the array, one value at a time.
2. For each value, compare the value with the next value.
3. If the value is higher than the next one, swap the values so that the highest value comes last.

Continue reading to fully understand the Bubble Sort algorithm and how to implement it yourself.

## Manual Run Through

Before we implement the Bubble Sort algorithm in a programming language, let's manually run through a short array only one time, just to get the idea.

**Step 1:** We start with an unsorted array.

```
[7, 12, 9, 11, 3]
```

**Step 2:** We look at the two first values. Does the lowest value come first? Yes, so we don't need to swap them.

```
[7, 12, 9, 11, 3]
```

**Step 3:** Take one step forward and look at values 12 and 9. Does the lowest value come first? No.

```
[7, 12, 9, 11, 3]
```

**Step 4:** So we need to swap them so that 9 comes first.

```
[7, 9, 12, 11, 3]
```

**Step 5:** Taking one step forward, looking at 12 and 11.

```
[7, 9, 12, 11, 3]
```

**Step 6:** We must swap so that 11 comes before 12.

```
[7, 9, 11, 12, 3]
```

**Step 8:** Swapping 12 and 3 so that 3 comes first.

[7, 9, 11, 3, 12]

Run the simulation below to see the 8 steps above animated:

Run Through Once

[ 7, 12, 9, 11, 3 ]

## Manual Run Through: What Happened?

We must understand what happened in this first run through to fully understand the algorithm, so that we can implement the algorithm in a programming language.

Can you see what happened to the highest value 12? It has bubbled up to the end of the array, where it belongs. But the rest of the array remains unsorted.

So the Bubble Sort algorithm must run through the array again, and again, and again, each time the next highest value bubbles up to its correct position. The sorting continues until the lowest value 3 is left at the start of the array. This means that we need to run through the array 4 times, to sort the array of 5 values.

And each time the algorithm runs through the array, the remaining unsorted part of the array becomes shorter.

This is how a full manual run through looks like:

Bubble Sort

[ 7, 12, 9, 11, 3 ]

We will now use what we have learned to implement the Bubble Sort algorithm in a programming language.

To implement the Bubble Sort algorithm in a programming language, we need:

1. An array with values to sort.
2. An inner loop that goes through the array and swaps values if the first value is higher than the next value. This loop must loop through one less value each time it runs.
3. An outer loop that controls how many times the inner loop must run. For an array with  $n$  values, this outer loop must run  $n-1$  times.

The resulting code looks like this:

## Example

```
my_array = [64, 34, 25, 12, 22, 11, 90, 5]

n = len(my_array)
for i in range(n-1):
    for j in range(n-i-1):
        if my_array[j] > my_array[j+1]:
            my_array[j], my_array[j+1] = my_array[j+1], my_array[j]

print("Sorted array:", my_array)
```

[Run Example »](#)

## Bubble Sort Improvement

The Bubble Sort algorithm can be improved a little bit more.

Imagine that the array is almost sorted already, with the lowest numbers at the start, like this for example:

```
my_array = [7, 3, 9, 12, 11]
```

finished sorted, and we can stop the algorithm, like this:

## Example

```
my_array = [7, 3, 9, 12, 11]

n = len(my_array)
for i in range(n-1):

    for j in range(n-i-1):
        if my_array[j] > my_array[j+1]:
            my_array[j], my_array[j+1] = my_array[j+1], my_array[j]

print("Sorted array:", my_array)
```

[Run Example »](#)

## Bubble Sort Time Complexity

For a general explanation of what time complexity is, visit [this page](#).

For a more thorough and detailed explanation of Bubble Sort time complexity, visit [this page](#).

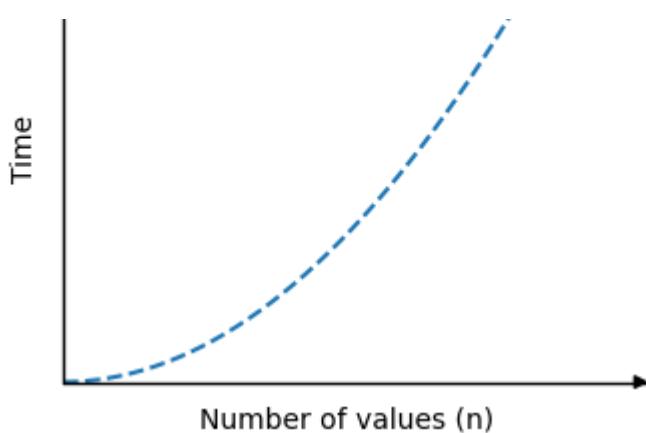
The Bubble Sort algorithm loops through every value in the array, comparing it to the value next to it. So for an array of  $n$  values, there must be  $n$  such comparisons in one loop.

And after one loop, the array is looped through again and again  $n$  times.

This means there are  $n \cdot n$  comparisons done in total, so the time complexity for Bubble Sort is:

$$\underline{\underline{O(n^2)}}$$

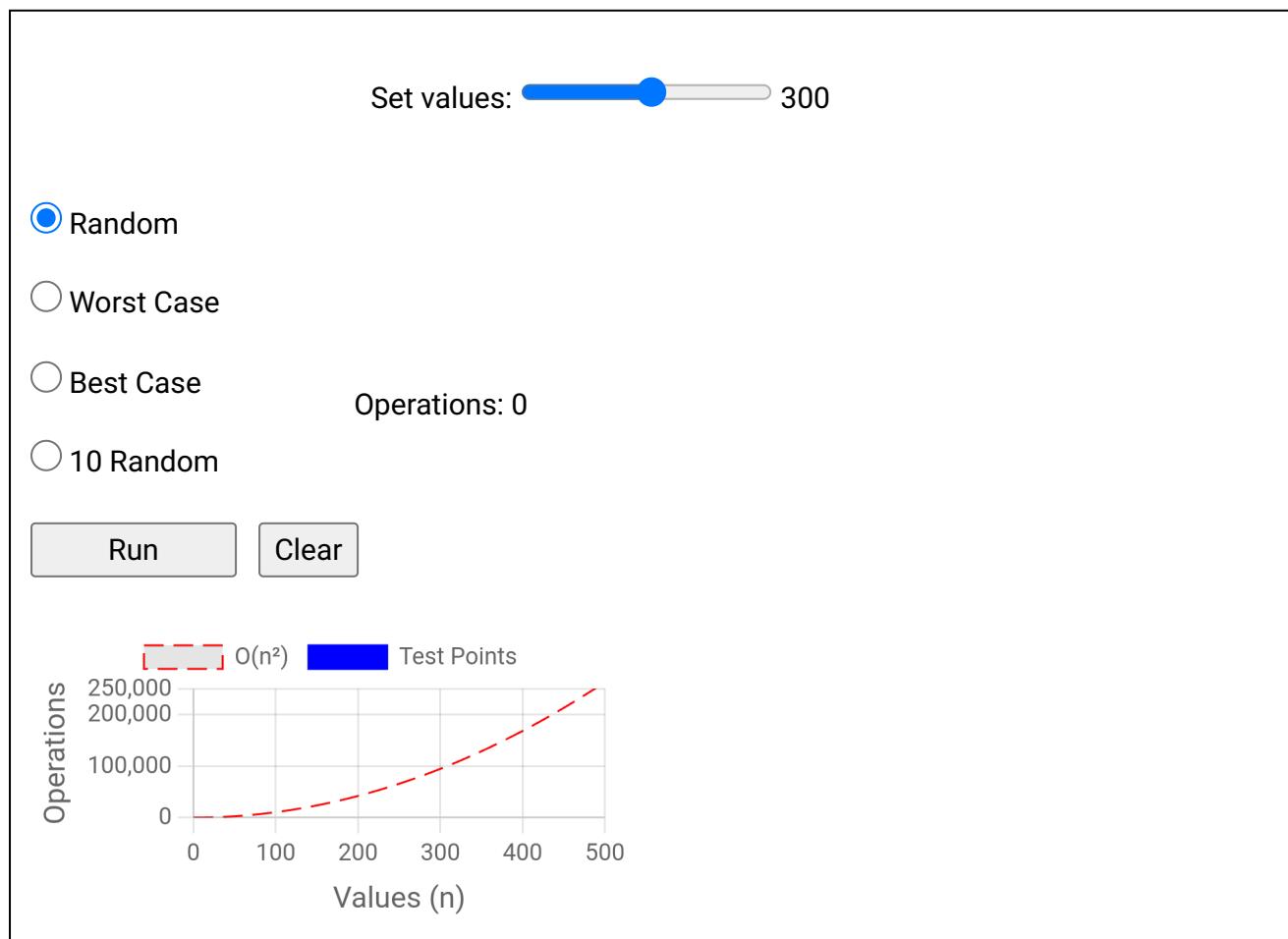
The graph describing the Bubble Sort time complexity looks like this:



As you can see, the run time increases really fast when the size of the array is increased.

Luckily there are sorting algorithms that are faster than this, like [Quicksort](#), that we will look at later.

You can simulate Bubble Sort below, where the red and dashed line is the theoretical time complexity  $O(n^2)$ . You can choose a number of values  $n$ , and run an actual Bubble Sort implementation where the operations are counted and the count is marked as a blue cross in the plot below. How does theory compare with practice?

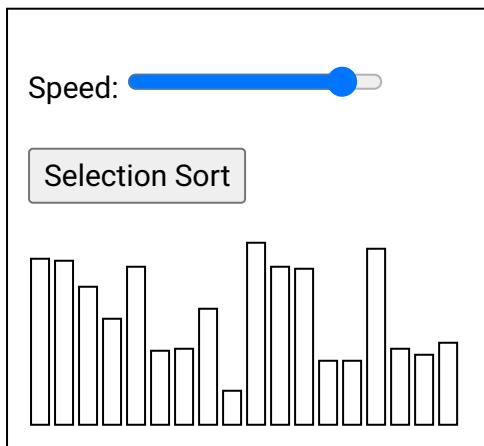


# DSA Selection Sort

[« Previous](#)[Next »](#)

## Selection Sort

The Selection Sort algorithm finds the lowest value in an array and moves it to the front of the array.



The algorithm looks through the array again and again, moving the next lowest values to the front, until the array is sorted.

### How it works:

1. Go through the array to find the lowest value.
2. Move the lowest value to the front of the unsorted part of the array.
3. Go through the array again as many times as there are values in the array.

# Manual Run Through

Before we implement the Selection Sort algorithm in a programming language, let's manually run through a short array only one time, just to get the idea.

**Step 1:** We start with an unsorted array.

```
[ 7, 12, 9, 11, 3]
```

**Step 2:** Go through the array, one value at a time. Which value is the lowest? 3, right?

```
[ 7, 12, 9, 11, 3]
```

**Step 3:** Move the lowest value 3 to the front of the array.

```
[ 3, 7, 12, 9, 11]
```

**Step 4:** Look through the rest of the values, starting with 7. 7 is the lowest value, and already at the front of the array, so we don't need to move it.

```
[ 3, 7, 12, 9, 11]
```

**Step 5:** Look through the rest of the array: 12, 9 and 11. 9 is the lowest value.

```
[ 3, 7, 12, 9, 11]
```

**Step 6:** Move 9 to the front.

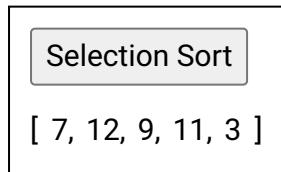
```
[ 3, 7, 9, 12, 11]
```

**Step 7:** Looking at 12 and 11, 11 is the lowest.

```
[ 3, 7, 9, 12, 11]
```

Finally, the array is sorted.

Run the simulation below to see the steps above animated:



## Manual Run Through: What Happened?

We must understand what happened above to fully understand the algorithm, so that we can implement the algorithm in a programming language.

Can you see what happened to the lowest value 3? In step 3, it has been moved to the start of the array, where it belongs, but at that step the rest of the array remains unsorted.

So the Selection Sort algorithm must run through the array again and again, each time the next lowest value is moved in front of the unsorted part of the array, to its correct position. The sorting continues until the highest value 12 is left at the end of the array. This means that we need to run through the array 4 times, to sort the array of 5 values.

And each time the algorithm runs through the array, the remaining unsorted part of the array becomes shorter.

We will now use what we have learned to implement the Selection Sort algorithm in a programming language.

## Selection Sort Implementation

To implement the Selection Sort algorithm in a programming language, we need:

1. An array with values to sort.
2. An inner loop that goes through the array, finds the lowest value, and moves it to the front of the array. This loop must loop through one less value each time it runs.

## Example

```
my_array = [64, 34, 25, 5, 22, 11, 90, 12]

n = len(my_array)
for i in range(n-1):
    min_index = i
    for j in range(i+1, n):
        if my_array[j] < my_array[min_index]:
            min_index = j
    min_value = my_array.pop(min_index)
    my_array.insert(i, min_value)

print("Sorted array:", my_array)
```

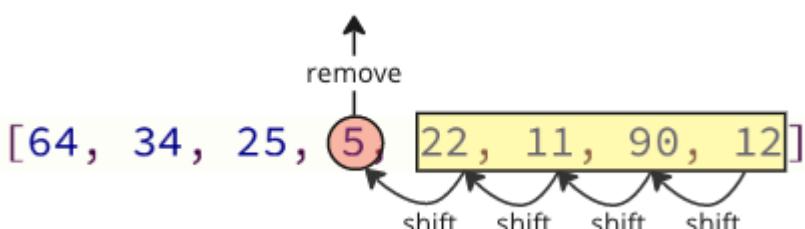
[Run Example »](#)

## Selection Sort Shifting Problem

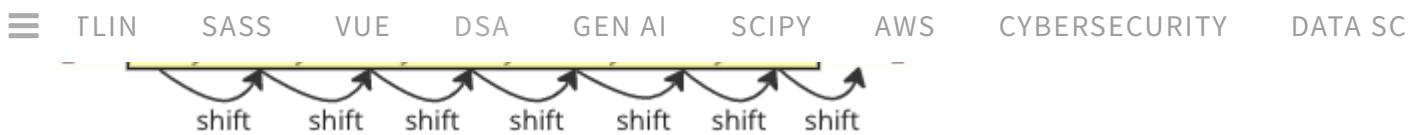
The Selection Sort algorithm can be improved a little bit more.

In the code above, the lowest value element is removed, and then inserted in front of the array.

Each time the next lowest value array element is removed, all following elements must be shifted one place down to make up for the removal.



These shifting operation takes a lot of time, and we are not even done yet! After the lowest value (5) is found and removed, it is inserted at the start of the array, causing all following values to shift one position up to make space for the new value, like the image below shows.



**Note:** You will not see these shifting operations happening in the code if you are using a high level programming language such as Python or Java, but the shifting operations are still happening in the background. Such shifting operations require extra time for the computer to do, which can be a problem.

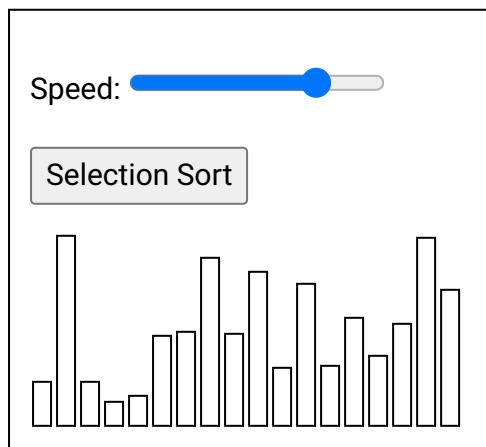
## Solution: Swap Values!

Instead of all the shifting, swap the lowest value (5) with the first value (64) like below.



We can swap values like the image above shows because the lowest value ends up in the correct position, and it does not matter where we put the other value we are swapping with, because it is not sorted yet.

Here is a simulation that shows how this improved Selection Sort with swapping works:



Here is an implementation of the improved Selection Sort, using swapping:

≡ TLIN SASS VUE DSA GEN AI SCIPY AWS CYBERSECURITY DATA SC

```
my_array = [64, 34, 25, 12, 22, 11, 90, 5]

n = len(my_array)
for i in range(n):
    min_index = i
    for j in range(i+1, n):
        if my_array[j] < my_array[min_index]:
            min_index = j

print("Sorted array:", my_array)
```

[Run Example »](#)

## Selection Sort Time Complexity

For a general explanation of what time complexity is, visit [this page](#).

For a more thorough and detailed explanation of Selection Sort time complexity, visit [this page](#).

Selection Sort sorts an array of  $n$  values.

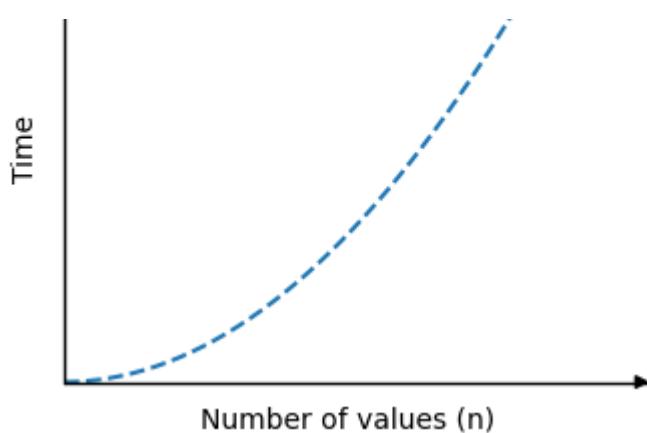
On average, about  $\frac{n}{2}$  elements are compared to find the lowest value in each loop.

And Selection Sort must run the loop to find the lowest value approximately  $n$  times.

We get time complexity:

$$O\left(\frac{n}{2} \cdot n\right) = \underline{\underline{O(n^2)}}$$

The time complexity for the Selection Sort algorithm can be displayed in a graph like this:

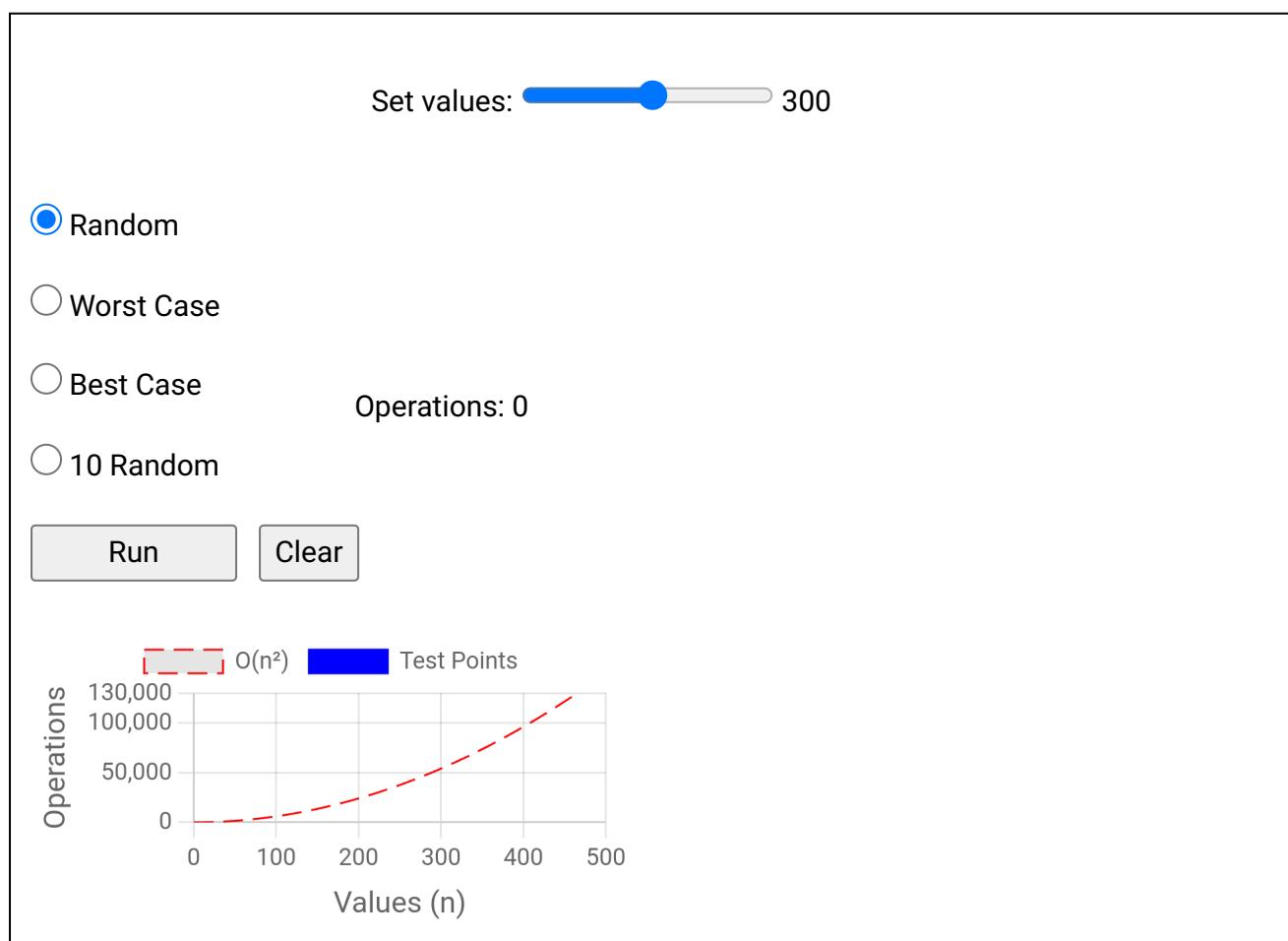


As you can see, the run time is the same as for Bubble Sort: The run time increases really fast when the size of the array is increased.

Run the simulation below for different sized arrays.

The red dashed line represents the theoretical time complexity  $O(n^2)$ .

Blue crosses appear when you run the simulation. The blue crosses show how many operations are needed to sort an array of a certain size.



The difference in best and worst case for Selection Sort is mainly the number of swaps. In the best case scenario Selection Sort does not have to swap any of the values because the array is already sorted. And in the worst case scenario, where the array already sorted, but in the wrong order, so Selection Sort must do as many swaps as there are values in array.

## DSA Exercises

### Test Yourself With Exercises

#### Exercise:

Using Selection Sort on this array:

[7, 12, 9, 11, 3]

To sort the values from left to right in an increasing (ascending) order.

What is the value of the LAST element after the first run through?

[Submit Answer »](#)

[Start the Exercise](#)

 Previous

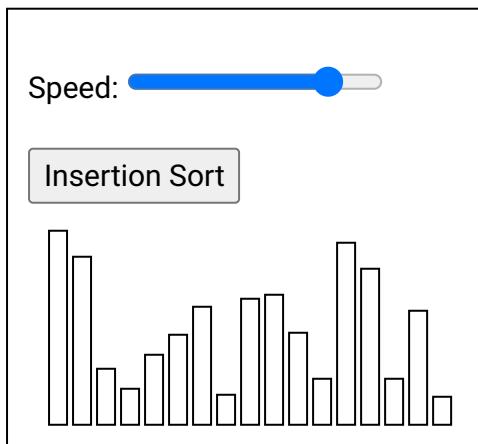
Next 

# DSA Insertion Sort

[« Previous](#)[Next »](#)

## Insertion Sort

The Insertion Sort algorithm uses one part of the array to hold the sorted values, and the other part of the array to hold values that are not sorted yet.



The algorithm takes one value at a time from the unsorted part of the array and puts it into the right place in the sorted part of the array, until the array is sorted.

### How it works:

1. Take the first value from the unsorted part of the array.
2. Move the value into the correct place in the sorted part of the array.
3. Go through the unsorted part of the array again as many times as there are values.

## MANUAL RUN THROUGH

Before we implement the Insertion Sort algorithm in a programming language, let's manually run through a short array, just to get the idea.

**Step 1:** We start with an unsorted array.

```
[ 7, 12, 9, 11, 3]
```

---

**Step 2:** We can consider the first value as the initial sorted part of the array. If it is just one value, it must be sorted, right?

```
[ 7, 12, 9, 11, 3]
```

---

**Step 3:** The next value 12 should now be moved into the correct position in the sorted part of the array. But 12 is higher than 7, so it is already in the correct position.

```
[ 7, 12, 9, 11, 3]
```

---

**Step 4:** Consider the next value 9.

```
[ 7, 12, 9, 11, 3]
```

---

**Step 5:** The value 9 must now be moved into the correct position inside the sorted part of the array, so we move 9 in between 7 and 12.

```
[ 7, 9, 12, 11, 3]
```

---

**Step 6:** The next value is 11.

```
[ 7, 9, 12, 11, 3]
```

---

**Step 7:** We move it in between 9 and 12 in the sorted part of the array.

**Step 8:** The last value to insert into the correct position is 3.

```
[ 7, 9, 11, 12, 3]
```

**Step 9:** We insert 3 in front of all other values because it is the lowest value.

```
[ 3, 7, 9, 11, 12]
```

Finally, the array is sorted.

Run the simulation below to see the steps above animated:

Insertion Sort

```
[ 7, 12, 9, 11, 3 ]
```

## Manual Run Through: What Happened?

We must understand what happened above to fully understand the algorithm, so that we can implement the algorithm in a programming language.

The first value is considered to be the initial sorted part of the array.

Every value after the first value must be compared to the values in the sorted part of the algorithm so that it can be inserted into the correct position.

The Insertion Sort Algorithm must run through the array 4 times, to sort the array of 5 values because we do not have to sort the first value.

And each time the algorithm runs through the array, the remaining unsorted part of the array becomes shorter.

We will now use what we have learned to implement the Insertion Sort algorithm in a programming language.

1. An array with values to sort.
2. An outer loop that picks a value to be sorted. For an array with  $n$  values, this outer loop skips the first value, and must run  $n - 1$  times.
3. An inner loop that goes through the sorted part of the array, to find where to insert the value.  
If the value to be sorted is at index  $i$ , the sorted part of the array starts at index 0 and ends at index  $i - 1$ .

The resulting code looks like this:

## Example

```
my_array = [64, 34, 25, 12, 22, 11, 90, 5]

n = len(my_array)
for i in range(1,n):
    insert_index = i
    current_value = my_array.pop(i)
    for j in range(i-1, -1, -1):
        if my_array[j] > current_value:
            insert_index = j
    my_array.insert(insert_index, current_value)

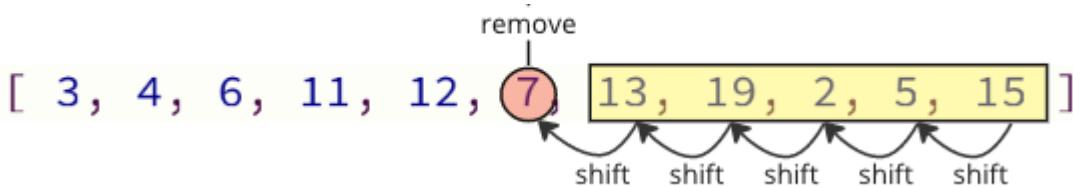
print("Sorted array:", my_array)
```

[Run Example »](#)

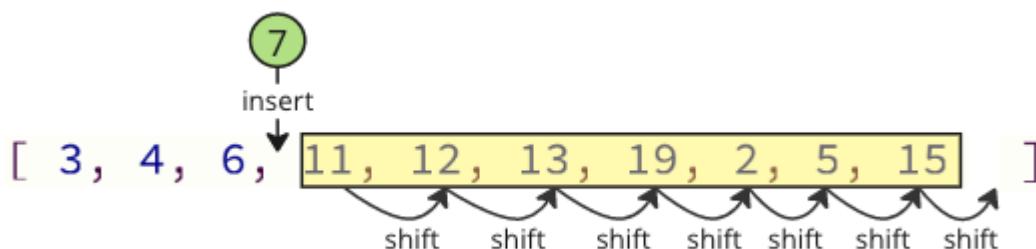
## Insertion Sort Improvement

Insertion Sort can be improved a little bit more.

The way the code above first removes a value and then inserts it somewhere else is intuitive. It is how you would do Insertion Sort physically with a hand of cards for example. If low value cards are sorted to the left, you pick up a new unsorted card, and insert it in the correct place between the other already sorted cards.



And when inserting the removed value into the array again, there are also many shift operations that must be done: all following elements must shift one position up to make place for the inserted value:

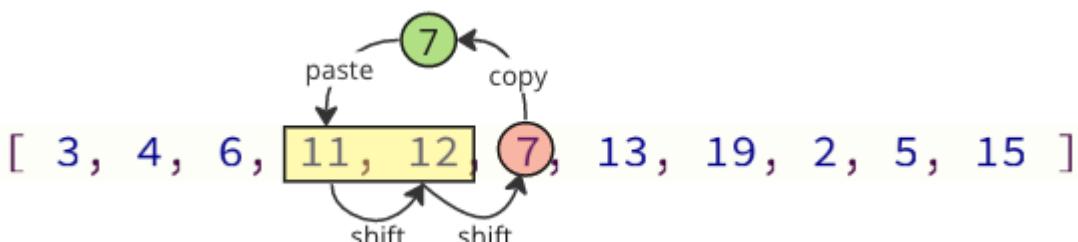


These shifting operation can take a lot of time, especially for an array with many elements.

**Note:** You will not see these shifting operations happening in the code if you are using a high level programming language such as Python or Java, but the shifting operations are still happening in the background. Such shifting operations require extra time for the computer to do, which can be a problem.

## Improved Solution

We can avoid most of these shift operations by only shifting the values necessary:



In the image above, first value 7 is copied, then values 11 and 12 are shifted one place up in the array, and at last value 7 is put where value 11 was before.

The number of shifting operations is reduced from 12 to 2 in this case.

```
my_array = [64, 34, 25, 12, 22, 11, 90, 5]

n = len(my_array)
for i in range(1,n):
    insert_index = i

    for j in range(i-1, -1, -1):
        if my_array[j] > current_value:

            insert_index = j

print("Sorted array:", my_array)
```

[Run Example »](#)

What is also done in the code above is to break out of the inner loop. That is because there is no need to continue comparing values when we have already found the correct place for the current value.

## Insertion Sort Time Complexity

For a general explanation of what time complexity is, visit [this page](#).

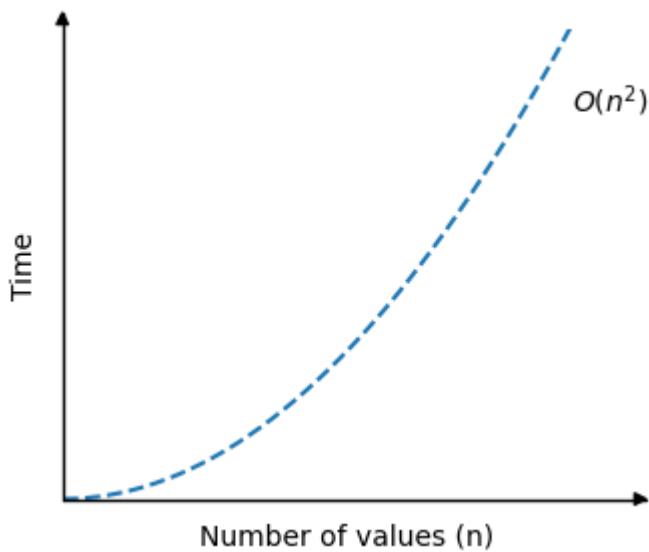
For a more thorough and detailed explanation of Insertion Sort time complexity, visit [this page](#).

Selection Sort sorts an array of  $n$  values.

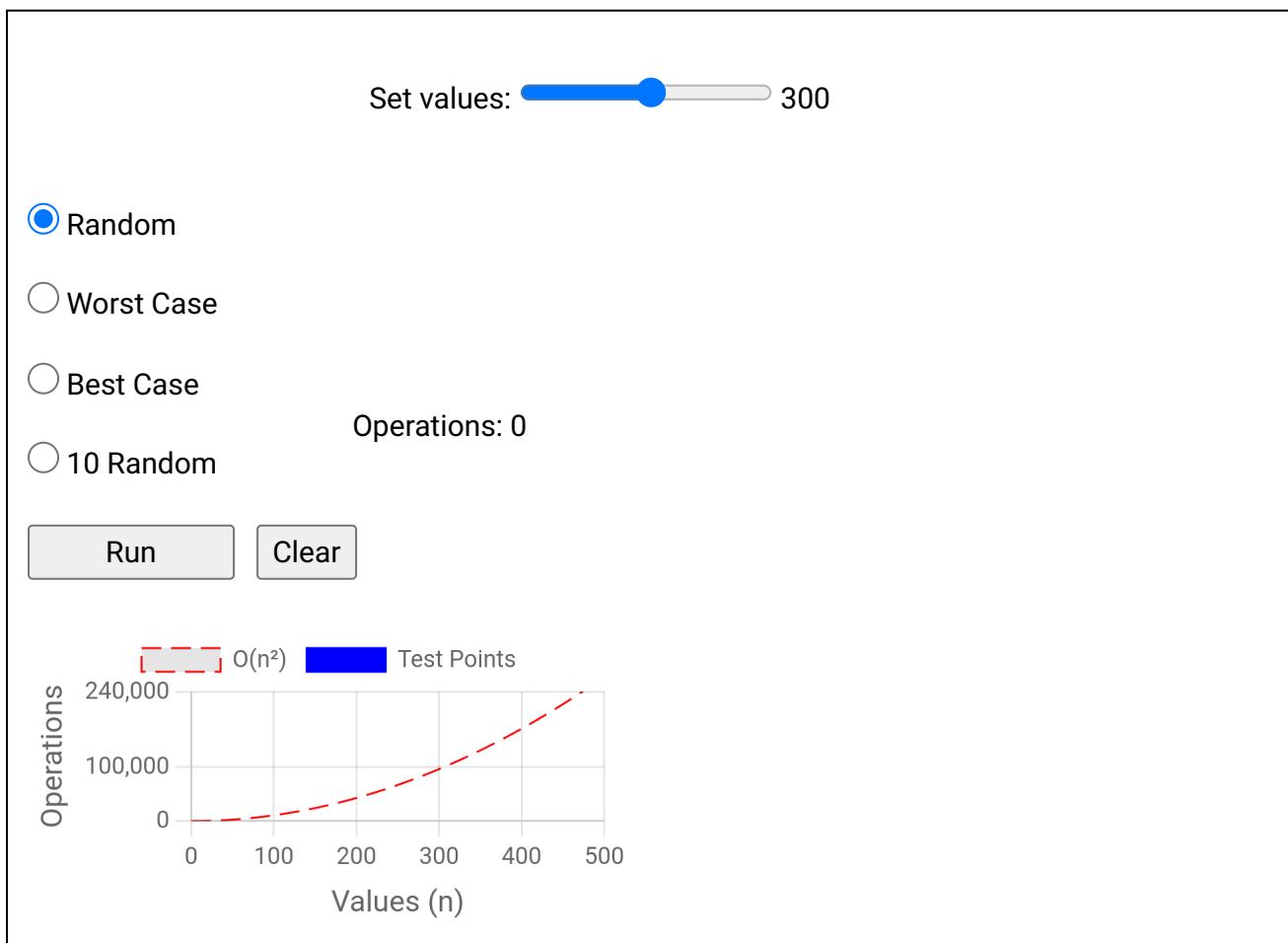
On average, each value must be compared to about  $\frac{n}{2}$  other values to find out where to insert it.

And Selection Sort must run the loop to insert a value in its correct place approximately  $n$  times.

We get time complexity for Insertion Sort:



Use the simulation below to see how the theoretical time complexity  $O(n^2)$  (red line) compares with the number of operations of actual Insertion Sorts.



For Insertion Sort, there is a big difference between best, average and worst case scenarios. You can see that by running the different simulations above.

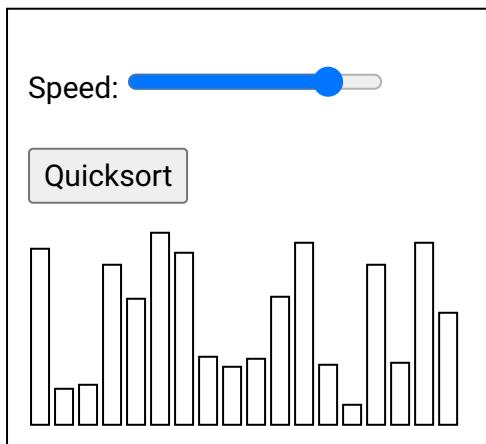
# DSA Quicksort

[« Previous](#)[Next »](#)

## Quicksort

As the name suggests, Quicksort is one of the fastest sorting algorithms.

The Quicksort algorithm takes an array of values, chooses one of the values as the 'pivot' element, and moves the other values so that lower values are on the left of the pivot element, and higher values are on the right of it.



In this tutorial the last element of the array is chosen to be the pivot element, but we could also have chosen the first element of the array, or any element in the array really.

Then, the Quicksort algorithm does the same operation recursively on the sub-arrays to the left and right side of the pivot element. This continues until the array is sorted.

**Recursion** is when a function calls itself.

sorted.

The algorithm can be described like this:

### How it works:

1. Choose a value in the array to be the pivot element.
2. Order the rest of the array so that lower values than the pivot element are on the left, and higher values are on the right.
3. Swap the pivot element with the first element of the higher values so that the pivot element lands in between the lower and higher values.
4. Do the same operations (recursively) for the sub-arrays on the left and right side of the pivot element.

Continue reading to fully understand the Quicksort algorithm and how to implement it yourself.

## Manual Run Through

Before we implement the Quicksort algorithm in a programming language, let's manually run through a short array, just to get the idea.

**Step 1:** We start with an unsorted array.

```
[ 11, 9, 12, 7, 3]
```

**Step 2:** We choose the last value 3 as the pivot element.

```
[ 11, 9, 12, 7, 3]
```

**Step 3:** The rest of the values in the array are all lower than 3, and must be on the right side of 3. Swap 3 with 11.

**Step 4:** value 3 is now in the correct position. we need to sort the values to the right of 3. we choose the last value 11 as the new pivot element.

[ 3, 9, 12, 7, 11 ]

**Step 5:** The value 7 must be to the left of pivot value 11, and 12 must be to the right of it. Move 7 and 12.

[ 3, 9, 7, 12, 11 ]

**Step 6:** Swap 11 with 12 so that lower values 9 and 7 are on the left side of 11, and 12 is on the right side.

[ 3, 9, 7, 11, 12 ]

**Step 7:** 11 and 12 are in the correct positions. We choose 7 as the pivot element in sub-array [ 9, 7], to the left of 11.

[ 3, 9, 7, 11, 12 ]

**Step 8:** We must swap 9 with 7.

[ 3, 7, 9, 11, 12 ]

And now, the array is sorted.

Run the simulation below to see the steps above animated:

Quicksort

happened above in more detail.

We have already seen that last value of the array is chosen as the pivot element, and the rest of the values are arranged so that the values lower than the pivot value are to the left, and the higher values are to the right.

After that, the pivot element is swapped with the first of the higher values. This splits the original array in two, with the pivot element in between the lower and the higher values.

Now we need to do the same as above with the sub-arrays on the left and right side of the old pivot element. And if a sub-array has length 0 or 1, we consider it finished sorted.

To sum up, the Quicksort algorithm makes the sub-arrays become shorter and shorter until array is sorted.

## Quicksort Implementation

To write a 'quickSort' method that splits the array into shorter and shorter sub-arrays we use recursion. This means that the 'quickSort' method must call itself with the new sub-arrays to the left and right of the pivot element. Read more about recursion [here](#).

To implement the Quicksort algorithm in a programming language, we need:

1. An array with values to sort.
2. A `quickSort` method that calls itself (recursion) if the sub-array has a size larger than 1.
3. A `partition` method that receives a sub-array, moves values around, swaps the pivot element into the sub-array and returns the index where the next split in sub-arrays happens.

The resulting code looks like this:

## Example

```
def partition(array, low, high):
    pivot = array[high]
    i = low - 1

    for j in range(low, high):
        if array[j] <= pivot:
            i += 1
            array[i], array[j] = array[j], array[i]
```

```
def quicksort(array, low=0, high=None):
    if high is None:
        high = len(array) - 1

    if low < high:
        pivot_index = partition(array, low, high)
        quicksort(array, low, pivot_index-1)
        quicksort(array, pivot_index+1, high)

my_array = [64, 34, 25, 12, 22, 11, 90, 5]
quicksort(my_array)
print("Sorted array:", my_array)
```

[Run Example »](#)

## Quicksort Time Complexity

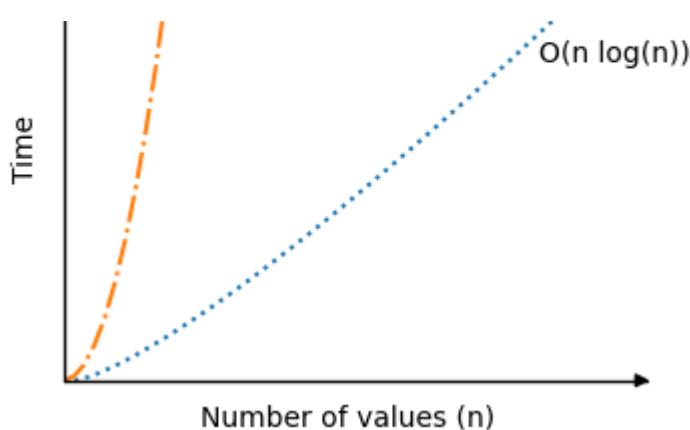
For a general explanation of what time complexity is, visit [this page](#).

For a more thorough and detailed explanation of Quicksort time complexity, visit [this page](#).

The worst case scenario for Quicksort is  $O(n^2)$ . This is when the pivot element is either the highest or lowest value in every sub-array, which leads to a lot of recursive calls. With our implementation above, this happens when the array is already sorted.

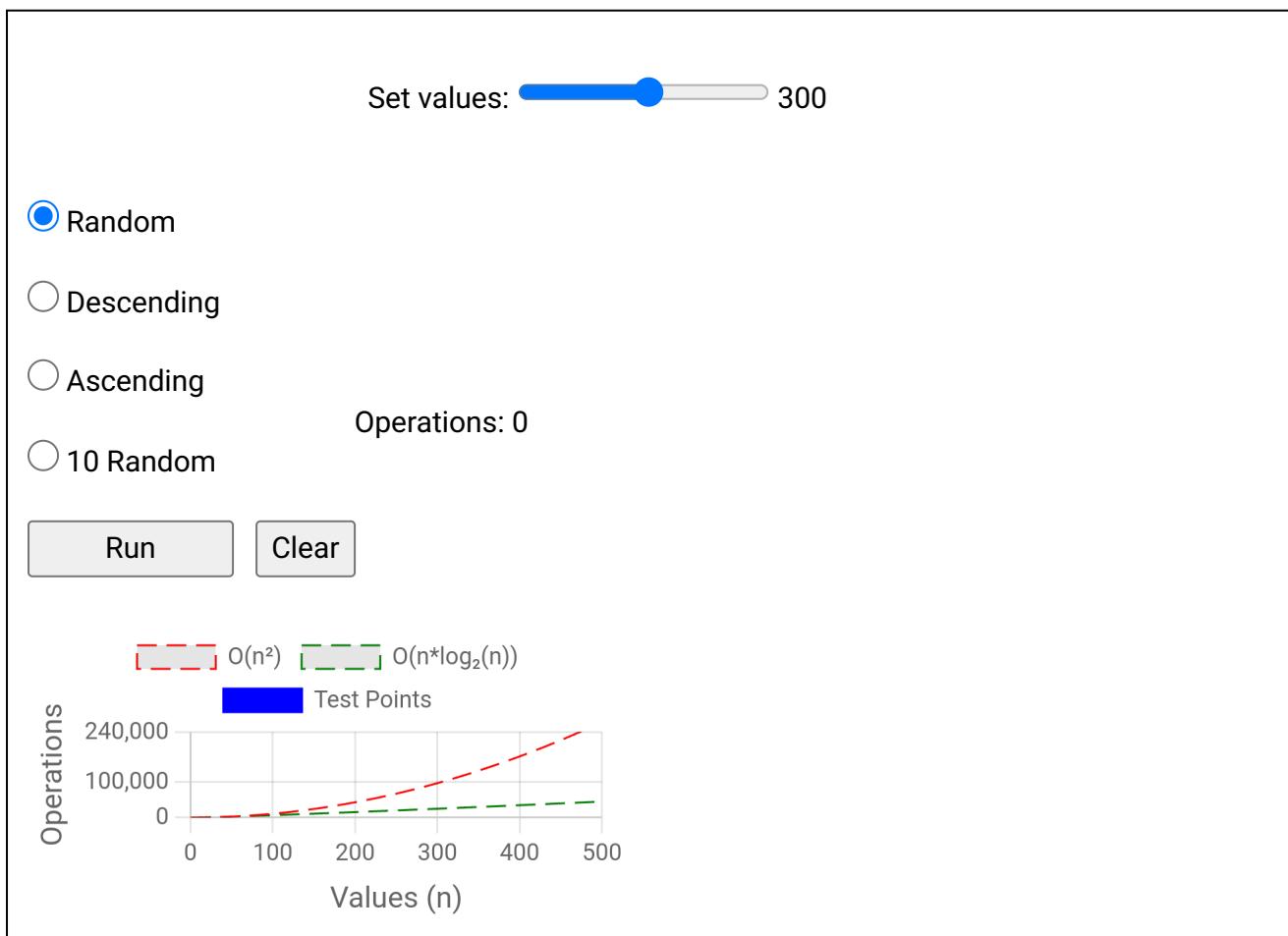
But on average, the time complexity for Quicksort is actually just  $O(n \log n)$ , which is a lot better than for the previous sorting algorithms we have looked at. That is why Quicksort is so popular.

Below you can see the significant improvement in time complexity for Quicksort in an average scenario  $O(n \log n)$ , compared to the previous sorting algorithms Bubble, Selection and Insertion Sort with time complexity  $O(n^2)$ :



The recursion part of the Quicksort algorithm is actually a reason why the average sorting scenario is so fast, because for good picks of the pivot element, the array will be split in half somewhat evenly each time the algorithm calls itself. So the number of recursive calls do not double, even if the number of values  $n$  double.

Run Quicksort on different kinds of arrays with different number of values in the simulation below:



## DSA Exercises

# Exercise:

Complete the code for the Quicksort algorithm.

```
def partition(array, low, high):
    pivot = array[high]
    i = low - 1

    for j in range(low, high):
        if array[j] <= pivot:
            i += 1
            array[i], array[j] = array[j], array[i]

    array[i+1], array[high] = array[high], array[i+1]
    return i+1

def quicksort(array, low=0, high=None):
    if high is None:
        high = len(array) - 1

    if low < high:
        pivot_index = partition(array, low, high)
        quicksort(array, low, pivot_index-1)
        quicksort(array, pivot_index+1, high)

my_array = [64, 34, 25, 12, 22, 11, 90, 5]
quicksort(my_array)
print("Sorted array:", my_array)
```

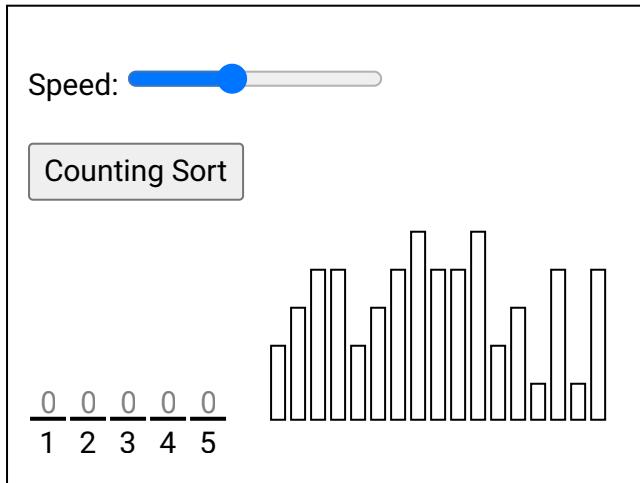
[Submit Answer »](#)

# DSA Counting Sort

[« Previous](#)[Next »](#)

## Counting Sort

The Counting Sort algorithm sorts an array by counting the number of times each value occurs.



Run the simulation to see how 17 integer values from 1 till 5 are sorted using Counting Sort.

Counting Sort does not compare values like the previous sorting algorithms we have looked at, and only works on non negative integers.

Furthermore, Counting Sort is fast when the range of possible values  $\{k\}$  is smaller than the number of values  $\{n\}$ .

### How it works:

1. Create a new array for counting how many there are of the different values.
2. Go through the array that needs to be sorted.

# Conditions for Counting Sort

These are the reasons why Counting Sort is said to only work for a limited range of non-negative integer values:

- **Integer values:** Counting Sort relies on counting occurrences of distinct values, so they must be integers. With integers, each value fits with an index (for non negative values), and there is a limited number of different values, so that the number of possible different values  $\backslash(k\backslash)$  is not too big compared to the number of values  $\backslash(n\backslash)$ .
- **Non negative values:** Counting Sort is usually implemented by creating an array for counting. When the algorithm goes through the values to be sorted, value  $x$  is counted by increasing the counting array value at index  $x$ . If we tried sorting negative values, we would get in trouble with sorting value -3, because index -3 would be outside the counting array.
- **Limited range of values:** If the number of possible different values to be sorted  $\backslash(k\backslash)$  is larger than the number of values to be sorted  $\backslash(n\backslash)$ , the counting array we need for sorting will be larger than the original array we have that needs sorting, and the algorithm becomes ineffective.

# Manual Run Through

Before we implement the Counting Sort algorithm in a programming language, let's manually run through a short array, just to get the idea.

**Step 1:** We start with an unsorted array.

```
myArray = [ 2, 3, 0, 2, 3, 2]
```

**Step 2:** We create another array for counting how many there are of each value. The array has 4 elements, to hold values 0 through 3.

```
myArray = [ 2, 3, 0, 2, 3, 2]
countArray = [ 0, 0, 0, 0]
```

```
countArray = [ 0, 0, 1, 0]
```

**Step 4:** After counting a value, we can remove it, and count the next value, which is 3.

```
myArray = [ 3, 0, 2, 3, 2]
countArray = [ 0, 0, 1, 1]
```

**Step 5:** The next value we count is 0, so we increment index 0 in the counting array.

```
myArray = [ 0, 2, 3, 2]
countArray = [ 1, 0, 1, 1]
```

**Step 6:** We continue like this until all values are counted.

```
myArray = []
countArray = [ 1, 0, 3, 2]
```

**Step 7:** Now we will recreate the elements from the initial array, and we will do it so that the elements are ordered lowest to highest.

The first element in the counting array tells us that we have 1 element with value 0. So we push 1 element with value 0 into the array, and we decrease the element at index 0 in the counting array with 1.

```
myArray = [ 0]
countArray = [ 0, 0, 3, 2]
```

**Step 8:** From the counting array we see that we do not need to create any elements with value 1.

```
myArray = [ 0]
countArray = [ 0, 0, 3, 2]
```

**Step 9:** We push 3 elements with value 2 into the end of the array. And as we create these elements we also decrease the counting array at index 2.

**Step 10:** At last we must add 2 elements with value 3 at the end of the array.

```
myArray = [0, 2, 2, 2, 3, 3]
countArray = [ 0, 0, 0, 0]
```

Finally! The array is sorted.

Run the simulation below to see the steps above animated:

#### Counting Sort

```
myArray = [ 2, 3, 0, 2, 3, 2 ]
countArray = [ 0, 0, 0, 0 ]
```

## Manual Run Through: What Happened?

Before we implement the algorithm in a programming language we need to go through what happened above in more detail.

We have seen that the Counting Sort algorithm works in two steps:

1. Each value gets counted by incrementing at the correct index in the counting array. After a value is counted, it is removed.
2. The values are recreated in the right order by using the count, and the index of the count, from the counting array.

With this in mind, we can start implementing the algorithm using Python.

## Counting Sort Implementation

To implement the Counting Sort algorithm in a programming language, we need:

1. An array with values to sort.
2. A 'countingSort' method that receives an array of integers.
3. An array inside the method to keep count of the values.
4. A loop inside the method that counts and removes values, by incrementing elements in the counting array.

array can be created with the correct size. For example, if the highest value is 5, the counting array must be 6 elements in total, to be able count all possible non negative integers 0, 1, 2, 3, 4 and 5.

The resulting code looks like this:

## Example

```
def countingSort(arr):
    max_val = max(arr)
    count = [0] * (max_val + 1)

    while len(arr) > 0:
        num = arr.pop(0)
        count[num] += 1

    for i in range(len(count)):
        while count[i] > 0:
            arr.append(i)
            count[i] -= 1

    return arr

unsortedArr = [4, 2, 2, 6, 3, 3, 1, 6, 5, 2, 3]
sortedArr = countingSort(unsortedArr)
print("Sorted array:", sortedArr)
```

[Run Example »](#)

## Counting Sort Time Complexity

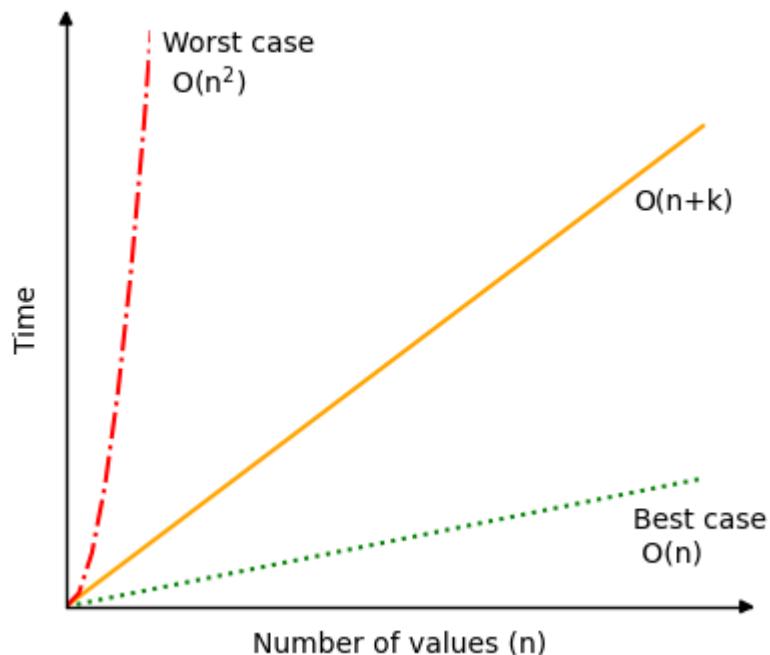
For a general explanation of what time complexity is, visit [this page](#).

For a more thorough and detailed explanation of Insertion Sort time complexity, visit [this page](#).

How fast the Counting Sort algorithm runs depends on both the range of possible values  $\backslash(k\backslash)$  and the number of values  $\backslash(n\backslash)$ .

But in a worst case scenario, the range of possible different values  $\backslash(k\backslash)$  is very big compared to the number of values  $\backslash(n\backslash)$  and Counting Sort can have time complexity  $\backslash(O(n^2)\backslash)$  or even worse.

The plot below shows how much the time complexity for Counting Sort can vary.



As you can see, it is important to consider the range of values compared to the number of values to be sorted before choosing Counting Sort as your algorithm. Also, as mentioned at the top of the page, keep in mind that Counting Sort only works for non negative integer values.

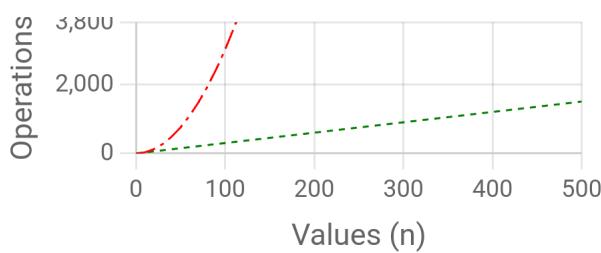
Run different simulations of Counting Sort to see how the number of operations falls between the worst case scenario  $\backslash(O(n^2)\backslash)$  (red line) and best case scenario  $\backslash(O(n)\backslash)$  (green line).

Set values (n):  300

Range (k), from 0 to:  1000

Random  
 Descending  
 Ascending  
 10 Random

Operations: 0



As mentioned previously: if the numbers to be sorted varies a lot in value (large  $\backslash(k\backslash)$ ), and there are few numbers to sort (small  $\backslash(n\backslash)$ ), the Counting Sort algorithm is not effective.

If we hold  $\backslash(n\backslash)$  and  $\backslash(k\backslash)$  fixed, the "Random", "Descending" and "Ascending" alternatives in the simulation above results in the same number of operations. This is because the same thing happens in all three cases: A counting array is set up, the numbers are counted, and the new sorted array is created.

## DSA Exercises

### Test Yourself With Exercises

#### Exercise:

Using Counting Sort on this array:

[1, 0, 5, 3, 3, 1, 3, 3, 4, 4]

How does the counting array look like?

[ , , , 4, 2, 1]

[Submit Answer »](#)

# DSA Radix Sort

[« Previous](#)[Next »](#)

## Radix Sort

The Radix Sort algorithm sorts an array by individual digits, starting with the least significant digit (the rightmost one).

Click the button to do Radix Sort, one step (digit) at a time.

Step

1 1 4
3 6 7
2 6 5
5 8 2
1 2 5
4 9 1
1 3 6
5 3 7
2 0 6
5 0 4

The radix (or base) is the number of unique digits in a number system. In the decimal system we normally use, there are 10 different digits from 0 till 9.

Radix Sort uses the radix so that decimal values are put into 10 different buckets (or containers) corresponding to the digit that is in focus, then put back into the array before moving on to the next digit.

Radix Sort is a non comparative algorithm that only works with non negative integers.

## How it works:

1. Start with the least significant digit (rightmost digit).
2. Sort the values based on the digit in focus by first putting the values in the correct bucket based on the digit in focus, and then put them back into array in the correct order.
3. Move to the next digit, and sort again, like in the step above, until there are no digits left.

# Stable Sorting

Radix Sort must sort the elements in a stable way for the result to be sorted correctly.

A stable sorting algorithm is an algorithm that keeps the order of elements with the same value before and after the sorting. Let's say we have two elements "K" and "L", where "K" comes before "L", and they both have value "3". A sorting algorithm is considered stable if element "K" still comes before "L" after the array is sorted.

It makes little sense to talk about stable sorting algorithms for the previous algorithms we have looked at individually, because the result would be same if they are stable or not. But it is important for Radix Sort that the the sorting is done in a stable way because the elements are sorted by just one digit at a time.

So after sorting the elements on the least significant digit and moving to the next digit, it is important to not destroy the sorting work that has already been done on the previous digit position, and that is why we need to take care that Radix Sort does the sorting on each digit position in a stable way.

In the simulation below it is revealed how the underlying sorting into buckets is done. And to get a better understanding of how stable sorting works, you can also choose to sort in an unstable way, that will lead to an incorrect result. The sorting is made unstable by simply putting elements into buckets from the end of the array instead of from the start of the array.

### Radix Sort

0  
1  
2  
3  
4  
5  
6  
7  
8  
9

4 0 9
1 3 2
5 9 1
2 0 5
4 9 8
1 1 2
3 4 3
1 2 4
4 0 1
2 0 9

## Manual Run Through

Let's try to do the sorting manually, just to get an even better understanding of how Radix Sort works before actually implementing it in a programming language.

**Step 1:** We start with an unsorted array, and an empty array to fit values with corresponding radices 0 till 9.

```
myArray = [ 33, 45, 40, 25, 17, 24]
radixArray = [ [], [], [], [], [], [], [], [], [] ]
```

**Step 2:** We start sorting by focusing on the least significant digit.

```
myArray = [ 33, 45, 40, 25, 17, 24]
radixArray = [ [], [], [], [], [], [], [], [], [] ]
```

**Step 3:** Now we move the elements into the correct positions in the radix array according to the digit in focus. Elements are taken from the start of myArray and pushed into the correct position in the radixArray.

```
myArray = [ ]
radixArray = [ [40], [], [], [33], [24], [45, 25], [], [17], [], [] ]
```

☰ TLIN SASS VUE DSA GEN AI SCIPY AWS CYBERSECURITY DATA SC

```
radixArray = [ [], [], [], [], [], [], [], [], [], [] ]
```

**Step 5:** We move focus to the next digit. Notice that values 45 and 25 are still in the same order relative to each other as they were to start with, because we sort in a stable way.

```
myArray = [ 40, 33, 24, 45, 25, 17 ]
radixArray = [ [], [], [], [], [], [], [], [], [], [] ]
```

**Step 6:** We move elements into the radix array according to the focused digit.

```
myArray = []
radixArray = [ [], [17], [24, 25], [33], [40, 45], [], [], [], [], [] ]
```

**Step 7:** We move elements back into the start of myArray, from the back of radixArray.

```
myArray = [ 17, 24, 25, 33, 40, 45 ]
radixArray = [ [], [], [], [], [], [], [], [], [], [] ]
```

The sorting is finished!

Run the simulation below to see the steps above animated:

Radix Sort

```
myArray = [ 33, 45, 40, 25, 17, 24 ]
radixArray = [ [], [], [], [], [], [], [], [], [], [] ]
```

## Manual Run Through: What Happened?

We see that values are moved from the array and placed in the radix array according to the current radix in focus. And then the values are moved back into the array we want to sort.

We also see that the radix array needs to be two-dimensional so that more than one value on a specific radix, or index.

And, as mentioned earlier, we must move values between the two arrays in a way that keeps the order of values with the same radix in focus, so the sorting is stable.

## Radix Sort Implementation

To implement the Radix Sort algorithm we need:

1. An array with non negative integers that needs to be sorted.
2. A two dimensional array with index 0 to 9 to hold values with the current radix in focus.
3. A loop that takes values from the unsorted array and places them in the correct position in the two dimensional radix array.
4. A loop that puts values back into the initial array from the radix array.
5. An outer loop that runs as many times as there are digits in the highest value.

The resulting code looks like this:

## Example

```
1 | myArray = [170, 45, 75, 90, 802, 24, 2, 66]
2 | print("Original array:", myArray)
3 | radixArray = [[], [], [], [], [], [], [], [], []]
4 | maxVal = max(myArray)
5 | exp = 1
6 |
7 |
8 |         while len(myArray) > 0:
9 |             val = myArray.pop()
10 |
11 |                 radixIndex = val // exp
12 |                 radixArray[radixIndex].append(val)
13 |
14 |             for bucket in radixArray:
15 |                 while len(bucket) > 0:
16 |                     val = bucket.pop()
17 |                     myArray.append(val)
18 |
19 |
```

[Run Example »](#)

**On line 7**, we use floor division ("//") to divide the maximum value 802 by 1 the first time the while loop runs, the next time it is divided by 10, and the last time it is divided by 100. When using floor division "//", any number beyond the decimal point are disregarded, and an integer is returned.

**On line 11**, it is decided where to put a value in the radixArray based on its radix, or digit in focus. For example, the second time the outer while loop runs exp will be 10. Value 170 divided by 10 will be 17. The "%10" operation divides by 10 and returns what is left. In this case 17 is divided by 10 one time, and 7 is left. So value 170 is placed in index 7 in the radixArray.

## Radix Sort Using Other Sorting Algorithms

Radix Sort can actually be implemented together with any other sorting algorithm as long as it is stable. This means that when it comes down to sorting on a specific digit, any stable sorting algorithm will work, such as counting sort or bubble sort.

This is an implementation of Radix Sort that uses Bubble Sort to sort on the individual digits:

### Example

```
def radixSortWithBubbleSort(arr):
    max_val = max(arr)
    exp = 1

    while max_val // exp > 0:
        radixArray = [[],[],[],[],[],[],[],[],[],[]]

        for num in arr:
            radixIndex = (num // exp) % 10
```

```
i = 0
for bucket in radixArray:
    for num in bucket:
        arr[i] = num
        i += 1

exp *= 10

myArray = [170, 45, 75, 90, 802, 24, 2, 66]
print("Original array:", myArray)
radixSortWithBubbleSort(myArray)
print("Sorted array:", myArray)
```

[Run Example »](#)

## Radix Sort Time Complexity

For a general explanation of what time complexity is, visit [this page](#).

For a more thorough and detailed explanation of Radix Sort time complexity, visit [this page](#).

The time complexity for Radix Sort is:

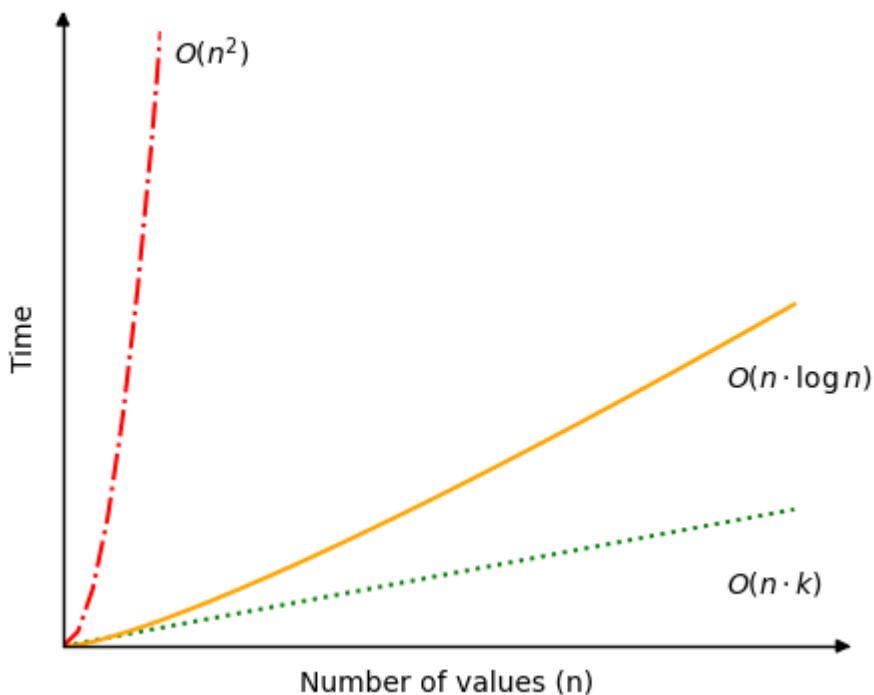
$$\underline{\underline{O(n \cdot k)}}$$

This means that Radix Sort depends both on the values that need to be sorted  $n$ , and the number of digits in the highest value  $k$ .

A best case scenario for Radix Sort is if there are lots of values to sort, but the values have few digits. For example if there are more than a million values to sort, and the highest value is 999, with just three digits. In such a case the time complexity  $O(n \cdot k)$  can be simplified to just  $O(n)$ .

A worst case scenario for Radix Sort would be if there are as many digits in the highest value as there are values to sort. This is perhaps not a common scenario, but the time complexity would be  $O(n^2)$  in this case.

The most average or common case is perhaps if the number of digits  $k$  is something like  $k(n) = \log n$ . If so, Radix Sort gets time complexity  $O(n \cdot \log n)$ . An example of such a case



Run different simulations of Radix Sort to see how the number of operations falls between the worst case scenario  $O(n^2)$  (red line) and best case scenario  $O(n)$  (green line).

Set values (n):  300

Digits (k):  4

Random

Descending

Ascending

10 Random

Operations: 0

Run Clear



The bars representing the different values are scaled to fit the window, so that it looks ok. This means that values with 7 digits look like they are just 5 times bigger than values with 2 digits, but in reality, values with 7 digits are actually 5000 times bigger than values with 2 digits!

If we hold  $n$  and  $k$  fixed, the "Random", "Descending" and "Ascending" alternatives in the simulation above results in the same number of operations. This is because the same thing happens in all three cases.

## DSA Exercises

### Test Yourself With Exercises

#### Exercise:

To sort an array with Radix Sort, what property must the sorting have for the sorting to be done properly?

Radix Sort must use a sorting algorithm.

[Submit Answer »](#)

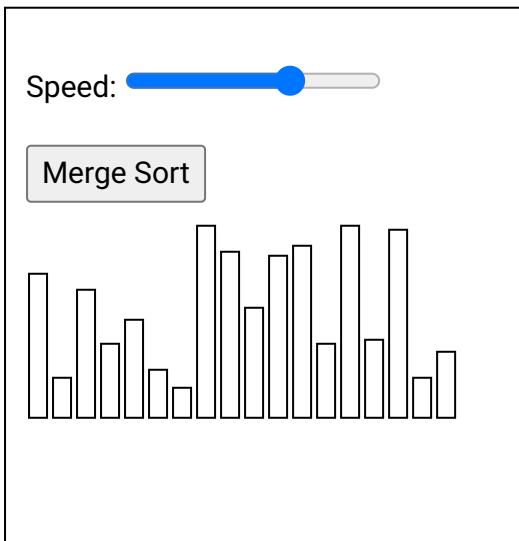
[Start the Exercise](#)

# DSA Merge Sort

[« Previous](#)[Next »](#)

## Merge Sort

The Merge Sort algorithm is a divide-and-conquer algorithm that sorts an array by first breaking it down into smaller arrays, and then building the array back together the correct way so that it is sorted.



**Divide:** The algorithm starts with breaking up the array into smaller and smaller pieces until one such sub-array only consists of one element.

**Conquer:** The algorithm merges the small pieces of the array back together by putting the lowest values first, resulting in a sorted array.

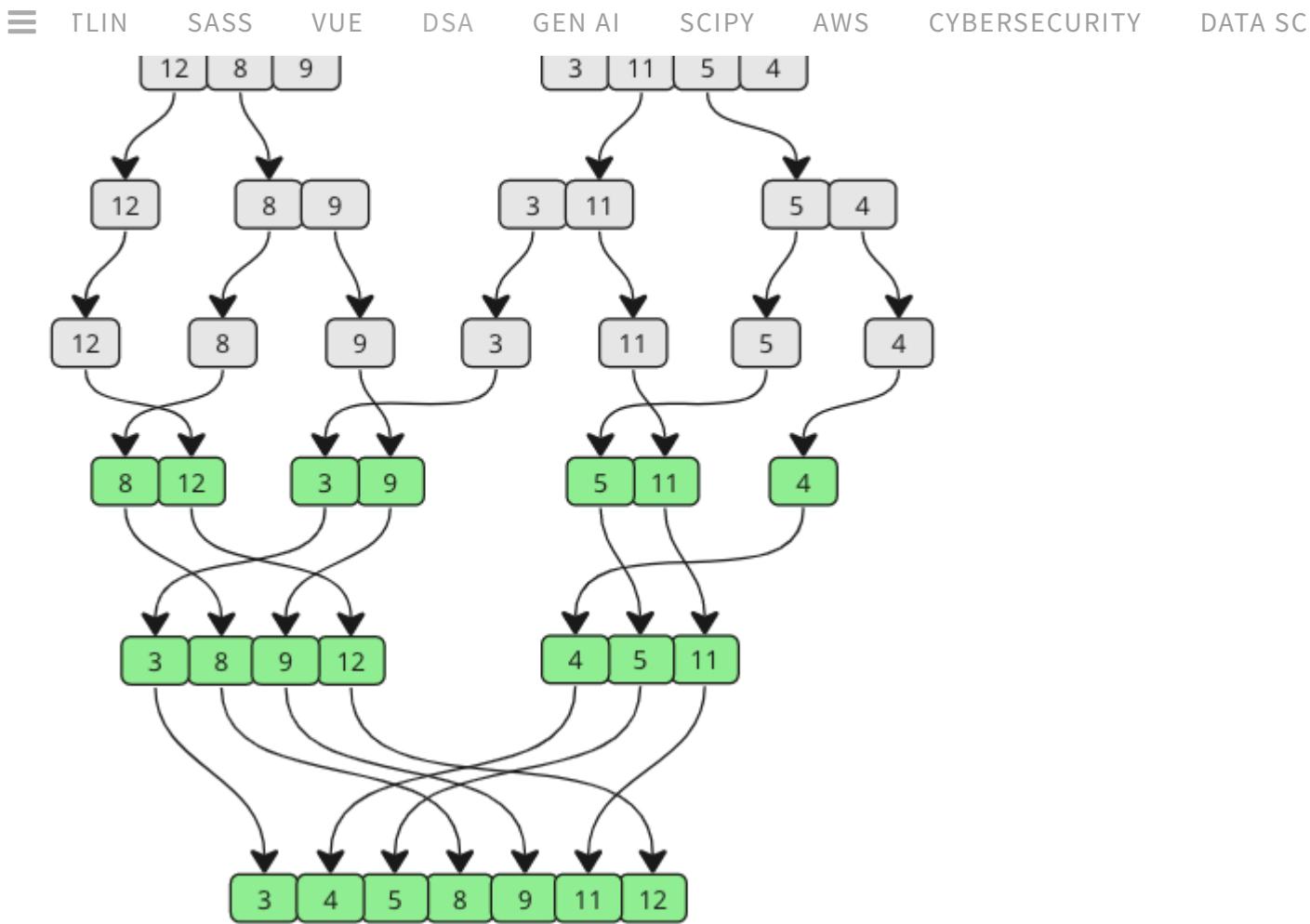
The breaking down and building up of the array to sort the array is done recursively.

The Merge Sort algorithm can be described like this:

### How it works:

1. Divide the unsorted array into two sub-arrays, half the size of the original.
2. Continue to divide the sub-arrays as long as the current piece of the array has more than one element.
3. Merge two sub-arrays together by always putting the lowest value first.
4. Keep merging until there are no sub-arrays left.

Take a look at the drawing below to see how Merge Sort works from a different perspective. As you can see, the array is split into smaller and smaller pieces until it is merged back together. And as the merging happens, values from each sub-array are compared so that the lowest value comes first.



## Manual Run Through

Let's try to do the sorting manually, just to get an even better understanding of how Merge Sort works before actually implementing it in a programming language.

**Step 1:** We start with an unsorted array, and we know that it splits in half until the sub-arrays only consist of one element. The Merge Sort function calls itself two times, once for each half of the array. That means that the first sub-array will split into the smallest pieces first.

```
[ 12, 8, 9, 3, 11, 5, 4]
[ 12, 8, 9] [ 3, 11, 5, 4]
[ 12] [ 8, 9] [ 3, 11, 5, 4]
[ 12] [ 8] [ 9] [ 3, 11, 5, 4]
```

**Step 2:** The splitting of the first sub-array is finished, and now it is time to merge. 8 and 9 are the first two elements to be merged. 8 is the lowest value, so that comes before 9 in the first merged sub-array.

**Step 3:** The next sub-arrays to be merged is [ 12] and [ 8, 9]. Values in both arrays are compared from the start. 8 is lower than 12, so 8 comes first, and 9 is also lower than 12.

```
[ 8, 9, 12] [ 3, 11, 5, 4]
```

**Step 4:** Now the second big sub-array is split recursively.

```
[ 8, 9, 12] [ 3, 11, 5, 4]  
[ 8, 9, 12] [ 3, 11] [ 5, 4]  
[ 8, 9, 12] [ 3] [ 11] [ 5, 4]
```

**Step 5:** 3 and 11 are merged back together in the same order as they are shown because 3 is lower than 11.

```
[ 8, 9, 12] [ 3, 11] [ 5, 4]
```

**Step 6:** Sub-array with values 5 and 4 is split, then merged so that 4 comes before 5.

```
[ 8, 9, 12] [ 3, 11] [ 5] [ 4]  
[ 8, 9, 12] [ 3, 11] [ 4, 5]
```

**Step 7:** The two sub-arrays on the right are merged. Comparisons are done to create elements in the new merged array:

1. 3 is lower than 4
2. 4 is lower than 11
3. 5 is lower than 11
4. 11 is the last remaining value

```
[ 8, 9, 12] [ 3, 4, 5, 11]
```

**Step 8:** The two last remaining sub-arrays are merged. Let's look at how the comparisons are done in more detail to create the new merged and finished sorted array:

3 is lower than 8:

**Step 9:** 4 is lower than 8:

```
Before [ 3, 8, 9, 12] [ 4, 5, 11]
```

```
After: [ 3, 4, 8, 9, 12] [ 5, 11]
```

**Step 10:** 5 is lower than 8:

```
Before [ 3, 4, 8, 9, 12] [ 5, 11]
```

```
After: [ 3, 4, 5, 8, 9, 12] [ 11]
```

**Step 11:** 8 and 9 are lower than 11:

```
Before [ 3, 4, 5, 8, 9, 12] [ 11]
```

```
After: [ 3, 4, 5, 8, 9, 12] [ 11]
```

**Step 12:** 11 is lower than 12:

```
Before [ 3, 4, 5, 8, 9, 12] [ 11]
```

```
After: [ 3, 4, 5, 8, 9, 11, 12]
```

The sorting is finished!

Run the simulation below to see the steps above animated:

Merge Sort

```
[ 12, 8, 9, 3, 11, 5, 4]
```

## Manual Run Through: What Happened?

We see that the algorithm has two stages: first splitting, then merging.

two, and then rounded down to get a value we call "mid". This "mid" value is used as an index for where to split the array.

After the array is split, the sorting function calls itself with each half, so that the array can be split again recursively. The splitting stops when a sub-array only consists of one element.

At the end of the Merge Sort function the sub-arrays are merged so that the sub-arrays are always sorted as the array is built back up. To merge two sub-arrays so that the result is sorted, the values of each sub-array are compared, and the lowest value is put into the merged array. After that the next value in each of the two sub-arrays are compared, putting the lowest one into the merged array.

## Merge Sort Implementation

To implement the Merge Sort algorithm we need:

1. An array with values that needs to be sorted.
2. A function that takes an array, splits it in two, and calls itself with each half of that array so that the arrays are split again and again recursively, until a sub-array only consist of one value.
3. Another function that merges the sub-arrays back together in a sorted way.

The resulting code looks like this:

## Example

```
1 | def mergeSort(arr):  
2 |     if len(arr) <= 1:  
3 |         return arr  
4 |  
5 |     mid = len(arr) // 2  
  
8 |  
9 |     sortedLeft = mergeSort(leftHalf)  
10|    sortedRight = mergeSort(rightHalf)  
11|  
12|    return merge(sortedLeft, sortedRight)  
13|  
14|
```

```
19     while i < len(left) and j < len(right):
20         if left[i] < right[j]:
21             result.append(left[i])
22             i += 1
23         else:
24             result.append(right[j])
25             j += 1

28     result.extend(right[j:])
29
30     return result
31
32 unsortedArr = [3, 7, 6, -10, 15, 23.5, 55, -13]
33 sortedArr = mergeSort(unsortedArr)
print("Sorted array:", sortedArr)
```

[Run Example »](#)

**On line 6**, arr[:mid] takes all values from the array up until, but not including, the value on index "mid".

**On line 7**, arr[mid:] takes all values from the array, starting at the value on index "mid" and all the next values.

**On lines 26-27**, the first part of the merging is done. At this point the values of the two sub-arrays are compared, and either the left sub-array or the right sub-array is empty, so the result array can just be filled with the remaining values from either the left or the right sub-array. These lines can be swapped, and the result will be the same.

## Merge Sort without Recursion

Since Merge Sort is a divide and conquer algorithm, recursion is the most intuitive code to use for implementation. The recursive implementation of Merge Sort is also perhaps easier to understand, and uses less code lines in general.

But Merge Sort can also be implemented without the use of recursion, so that there is no function calling itself.

```
def merge(left, right):
    result = []
    i = j = 0

    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

    result.extend(left[i:])
    result.extend(right[j:])

    return result

def mergeSort(arr):
    step = 1 # Starting with sub-arrays of length 1
    length = len(arr)

    return arr

unsortedArr = [3, 7, 6, -10, 15, 23.5, 55, -13]
sortedArr = mergeSort(unsortedArr)
print("Sorted array:", sortedArr)
```

You might notice that the merge functions are exactly the same in the two Merge Sort implementations above, but in the implementation right above here the while loop inside the `mergeSort` function is used to replace the recursion. The while loop does the splitting and merging of the array in place, and that makes the code a bit harder to understand.

To put it simply, the while loop inside the `mergeSort` function uses short step lengths to sort tiny pieces (sub-arrays) of the initial array using the `merge` function. Then the step length is increased to merge and sort larger pieces of the array until the whole array is sorted.

## Merge Sort Time Complexity

For a general explanation of what time complexity is, visit [this page](#).

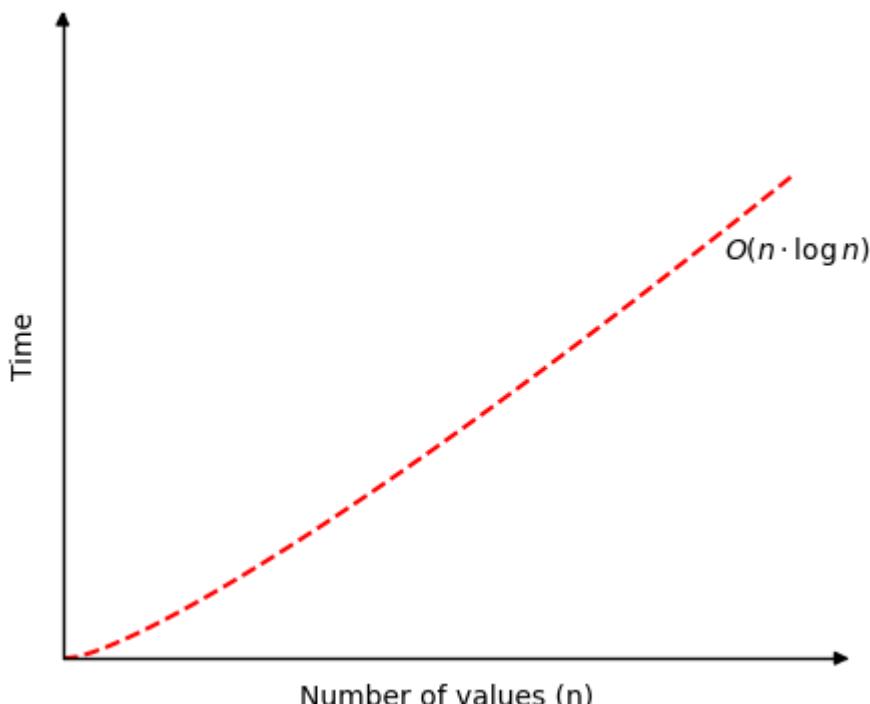
For a more thorough and detailed explanation of Merge Sort time complexity, visit [this page](#).

The time complexity for Merge Sort is

$$O(n \cdot \log n)$$

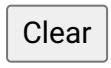
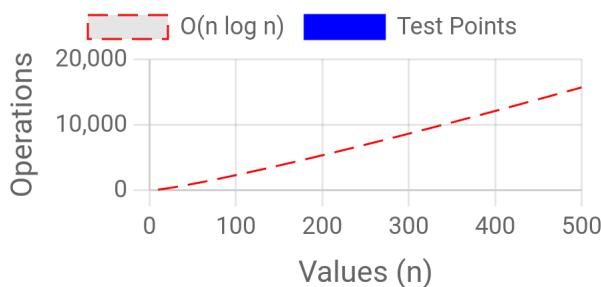
And the time complexity is pretty much the same for different kinds of arrays. The algorithm needs to split the array and merge it back together whether it is already sorted or completely shuffled.

The image below shows the time complexity for Merge Sort.



Set values:  300 Random Descending Ascending 10 Random

Operations: 0

 Run Clear

If we hold the number of values  $n$  fixed, the number of operations needed for the "Random", "Descending" and "Ascending" is almost the same.

Merge Sort performs almost the same every time because the array is split, and merged using comparison, both if the array is already sorted or not.

 PreviousNext 

W3schools Pathfinder  
Track your progress - it's free!

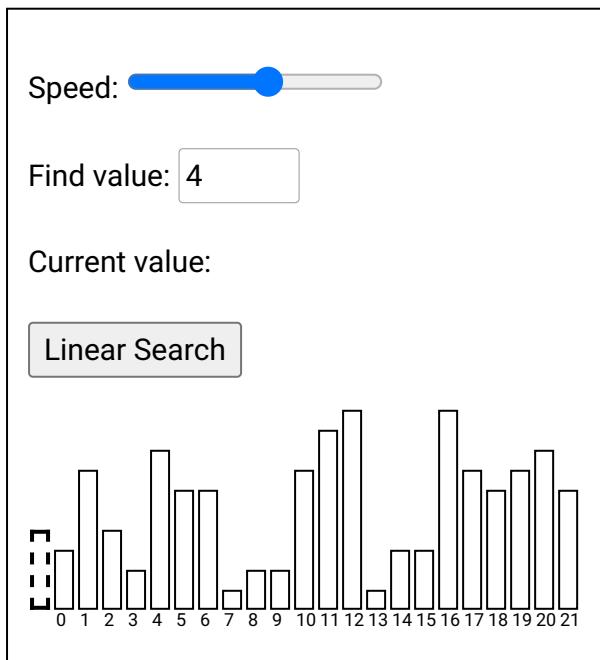
Sign Up Log in

# DSA Linear Search

[« Previous](#)[Next »](#)

## Linear Search

The Linear Search algorithm searches through an array and returns the index of the value it searches for.



Run the simulation above to see how the Linear Search algorithm works.

To see what happens when a value is not found, try to find value 5.

This algorithm is very simple and easy to understand and implement.

If the array is already sorted, it is better to use the much faster Binary Search algorithm that we will explore on the next page.

[Tutorials ▾](#)[Exercises ▾](#)[Services ▾](#)[Sign Up](#)[Log in](#)[☰](#) TLIN SASS VUE DSA GEN AI SCIPY AWS CYBERSECURITY DATA SC

## How it works:

1. Go through the array value by value from the start.
2. Compare each value to check if it is equal to the value we are looking for.
3. If the value is found, return the index of that value.
4. If the end of the array is reached and the value is not found, return -1 to indicate that the value was not found.

## Manual Run Through

Let's try to do the searching manually, just to get an even better understanding of how Linear Search works before actually implementing it in a programming language. We will search for value 11.

**Step 1:** We start with an array of random values.

```
[ 12, 8, 9, 11, 5, 11]
```

**Step 2:** We look at the first value in the array, is it equal to 11?

```
[ 12, 8, 9, 11, 5, 11]
```

**Step 3:** We move on to the next value at index 1, and compare it to 11 to see if it is equal.

```
[ 12, 8, 9, 11, 5, 11]
```

**Step 4:** We check the next value at index 2.

```
[ 12, 8, 9, 11, 5, 11]
```

**Step 5:** We move on to the next value at index 3. Is it equal to 11?

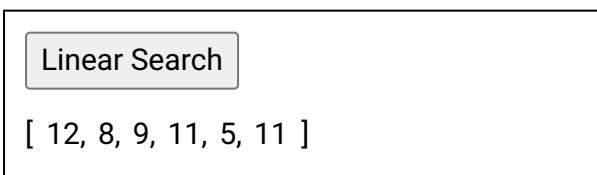
Value 11 is found at index 3.

Returning index position 3.

Linear Search is finished.

---

Run the simulation below to see the steps above animated:



## Manual Run Through: What Happened?

This algorithm is really straight forward.

Every value is checked from the start of the array to see if the value is equal to 11, the value we are trying to find.

When the value is found, the searching is stopped, and the index where the value is found is returned.

If the array is searched through without finding the value, -1 is returned.

---

## Linear Search Implementation

To implement the Linear Search algorithm we need:

1. An array with values to search through.
2. A target value to search for.
3. A loop that goes through the array from start to end.
4. An if-statement that compares the current value with the target value, and returns the current index if the target value is found.
5. After the loop, return -1, because at this point we know the target value has not been found.

The resulting code for Linear Search looks like this:

```
def linearSearch(arr, targetVal):
    for i in range(len(arr)):
        if arr[i] == targetVal:
            return i
    return -1

arr = [3, 7, 2, 9, 5]
targetVal = 9

result = linearSearch(arr, targetVal)

if result != -1:
    print("Value", targetVal, "found at index", result)
else:
    print("Value", targetVal, "not found")
```

[Run Example »](#)

## Linear Search Time Complexity

For a general explanation of what time complexity is, visit [this page](#).

For a more thorough and detailed explanation of Insertion Sort time complexity, visit [this page](#).

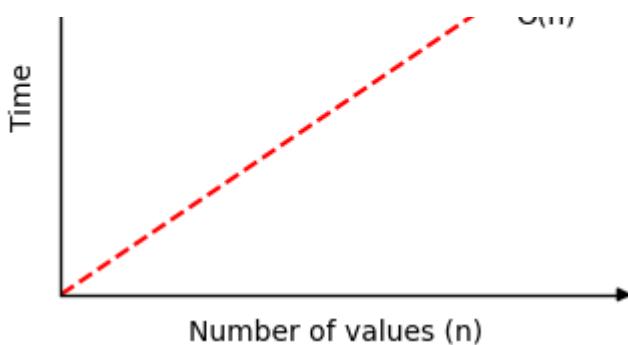
If Linear Search runs and finds the target value as the first array value in an array with  $n$  values, only one compare is needed.

But if Linear Search runs through the whole array of  $n$  values, without finding the target value,  $n$  compares are needed.

This means that time complexity for Linear Search is

$$O(n)$$

If we draw how much time Linear Search needs to find a value in an array of  $n$  values, we get this graph:



Run the simulation below for different number of values in an array, and see how many compares are needed for Linear Search to find a value in an array of  $n$  values:

Set values:  300

Random

Descending

Ascending

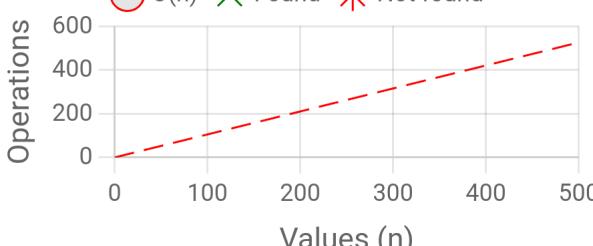
10 Random

Operations: 0

Run Clear

Operations (y-axis) vs Values (n) (x-axis)

Legend: ●  $O(n)$  ✗ Found \* Not found



Values (n)	Operations
0	0
100	~100
200	~200
300	~300
400	~400
500	~500

Choosing "Random", "Descending" or "Ascending" in the simulation above has no effect on how fast Linear Search is.

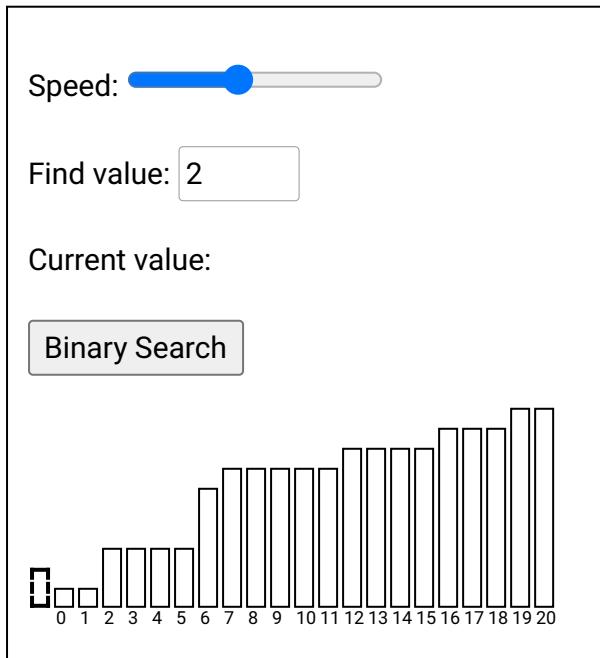
## DSA Exercises

# DSA Binary Search

[« Previous](#)[Next »](#)

## Binary Search

The Binary Search algorithm searches through an array and returns the index of the value it searches for.



Run the simulation to see how the Binary Search algorithm works.

To see what happens when a value is not found, try to find value 5.

Binary Search is much faster than Linear Search, but requires a sorted array to work.

The Binary Search algorithm works by checking the value in the center of the array. If the target value is lower, the next value to check is in the center of the left half of the array. This way of

area of the array is empty.

### How it works:

1. Check the value in the center of the array.
2. If the target value is lower, search the left half of the array. If the target value is higher, search the right half.
3. Continue step 1 and 2 for the new reduced part of the array until the target value is found or until the search area is empty.
4. If the value is found, return the target value index. If the target value is not found, return -1.

## Manual Run Through

Let's try to do the searching manually, just to get an even better understanding of how Binary Search works before actually implementing it in a programming language. We will search for value 11.

**Step 1:** We start with an array.

```
[ 2, 3, 7, 7, 11, 15, 25]
```

**Step 2:** The value in the middle of the array at index 3, is it equal to 11?

```
[ 2, 3, 7, 7, 11, 15, 25]
```

**Step 3:** 7 is less than 11, so we must search for 11 to the right of index 3. The values to the right of index 3 are [ 11, 15, 25]. The next value to check is the middle value 15, at index 5.

```
[ 2, 3, 7, 7, 11, 15, 25]
```

**Step 4:** 15 is higher than 11, so we must search to the left of index 5. We have already checked index 0-3, so index 4 is only value left to check.

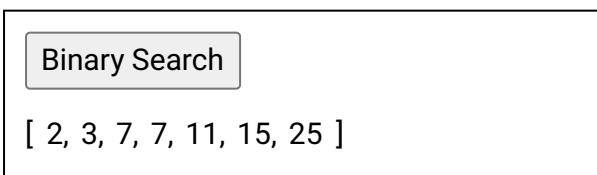
Value 11 is found at index 4.

Returning index position 4.

Binary Search is finished.

---

Run the simulation below to see the steps above animated:



## Manual Run Through: What Happened?

To start with, the algorithm has two variables "left" and "right".

"left" is 0 and represents the index of the first value in the array, and "right" is 6 and represents the index of the last value in the array.

$(left + right)/2 = (0 + 6)/2 = 3$  is the first index used to check if the middle value (7) is equal to the target value (11).

7 is lower than the target value 11, so in the next loop the search area must be limited to the right side of the middle value: [ 11, 15, 25], on index 4-6.

To limit the search area and find a new middle value, "left" is updated to index 4, "right" is still 6. 4 and 6 are the indexes for the first and last values in the new search area, the right side of the previous middle value. The new middle value index is

$$(left + right)/2 = (4 + 6)/2 = 10/2 = 5.$$

The new middle value on index 5 is checked: 15 is higher than 11, so if the target value 11 exists in the array it must be on the left side of index 5. The new search area is created by updating "right" from 6 to 4. Now both "left" and "right" is 4,  $(left + right)/2 = (4 + 4)/2 = 4$ , so there is only index 4 left to check. The target value 11 is found at index 4, so index 4 is returned.

In general, this is the way the Binary Search algorithm continues to halve the array search area until the target value is found.

# Binary Search Implementation

To implement the Binary Search algorithm we need:

1. An array with values to search through.
2. A target value to search for.
3. A loop that runs as long as left index is less than, or equal to, the right index.
4. An if-statement that compares the middle value with the target value, and returns the index if the target value is found.
5. An if-statement that checks if the target value is less than, or larger than, the middle value, and updates the "left" or "right" variables to narrow down the search area.
6. After the loop, return -1, because at this point we know the target value has not been found.

The resulting code for Binary Search looks like this:

## Example

```
def binarySearch(arr, targetVal):  
    left = 0  
    right = len(arr) - 1  
  
    while left <= right:  
        mid = (left + right) // 2  
  
        if arr[mid] == targetVal:  
            return mid  
  
        if arr[mid] < targetVal:  
            left = mid + 1  
        else:  
            right = mid - 1  
  
    return -1  
  
myArray = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]  
myTarget = 15  
  
result = binarySearch(myArray, myTarget)  
  
if result != -1:
```

[Run Example »](#)

# Binary Search Time Complexity

For a general explanation of what time complexity is, visit [this page](#).

For a more thorough and detailed explanation of Insertion Sort time complexity, visit [this page](#).

Each time Binary Search checks a new value to see if it is the target value, the search area is halved.

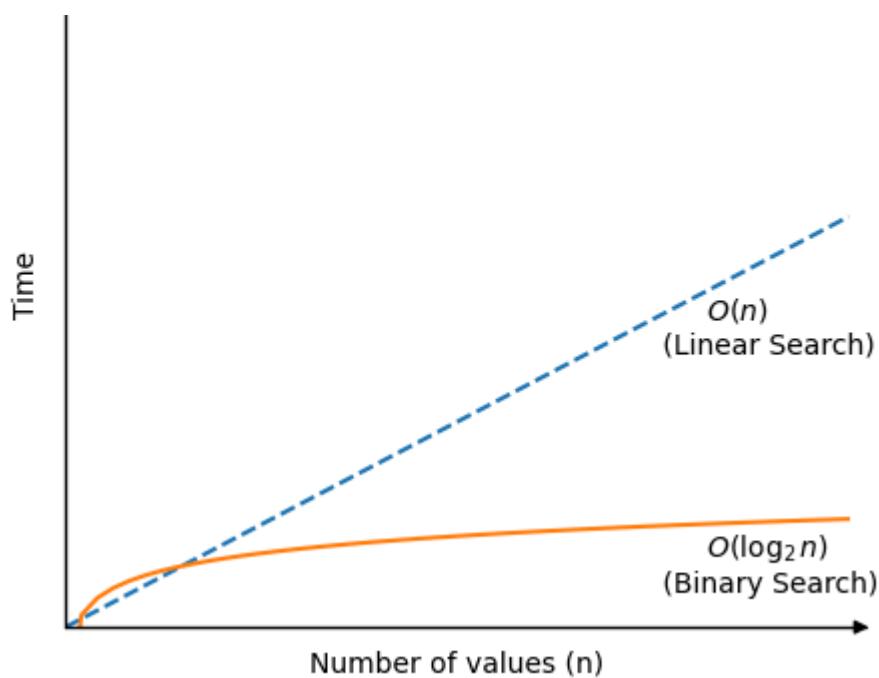
This means that even in the worst case scenario where Binary Search cannot find the target value, it still only needs  $\log_2 n$  comparisons to look through a sorted array of  $n$  values.

Time complexity for Binary Search is

$$O(\log_2 n)$$

**Note:** When writing time complexity using Big O notation we could also just have written  $O(\log n)$ , but  $O(\log_2 n)$  reminds us that the array search area is halved for every new comparison, which is the basic concept of Binary Search, so we will just keep the base 2 indication in this case.

If we draw how much time Binary Search needs to find a value in an array of  $n$  values, compared to Linear Search, we get this graph:



Run the Binary Search simulation below for different number of values  $n$  in an array, and see how many compares are needed for Binary Search to find the target value:

Set values:  300

Ascending

10 Ascending

Operations: 0

Run Clear

$O(\log_2 n)$ 
X Found
 \* Not Found

Operations

Values (n)	Operations
0	4
100	6
200	7.5
300	8.5
400	9.2
500	10

Values (n)

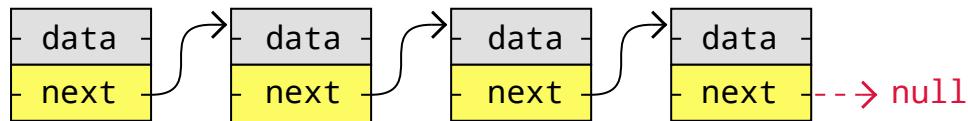
# DSA Linked Lists

[« Previous](#)[Next »](#)

A **Linked List** is, as the word implies, a list where the nodes are linked together. Each node contains data and a pointer. The way they are linked together is that each node points to where in the memory the next node is placed.

## Linked Lists

A linked list consists of nodes with some sort of data, and a pointer, or link, to the next node.



A big benefit with using linked lists is that nodes are stored wherever there is free space in memory, the nodes do not have to be stored contiguously right after each other like elements are stored in arrays. Another nice thing with linked lists is that when adding or removing nodes, the rest of the nodes in the list do not have to be shifted.

## Linked Lists vs Arrays

The easiest way to understand linked lists is perhaps by comparing linked lists with arrays.

other elements.

**Note:** How linked lists and arrays are stored in memory will be explained in more detail on [the next page](#).

The table below compares linked lists with arrays to give a better understanding of what linked lists are.

	Arrays	Linked Lists
<i>An existing data structure in the programming language</i>	Yes	No
<i>Fixed size in memory</i>	Yes	No
<i>Elements, or nodes, are stored right after each other in memory (contiguously)</i>	Yes	No
<i>Memory usage is low (each node only contains data, no links to other nodes)</i>	Yes	No
<i>Elements, or nodes, can be accessed directly (random access)</i>	Yes	No
<i>Elements, or nodes, can be inserted or deleted in constant time, no shifting operations in memory needed.</i>	No	Yes

To explain these differences in more detail, the next page will focus on how linked lists and arrays are stored in memory.

## DSA Exercises

Test Yourself With Exercises

Exercise:

# DSA Linked Lists in Memory

[« Previous](#)[Next »](#)

## Computer Memory

To explain what linked lists are, and how linked lists are different from arrays, we need to understand some basics about how computer memory works.

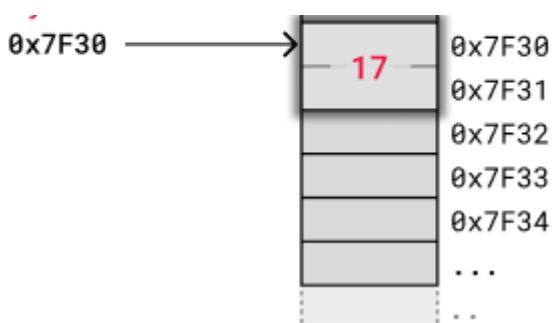
Computer memory is the storage your program uses when it is running. This is where your variables, arrays and linked lists are stored.

## Variables in Memory

Let's imagine that we want to store the integer "17" in a variable `myNumber`. For simplicity, let's assume the integer is stored as two bytes (16 bits), and the address in memory to `myNumber` is `0x7F30`.

`0x7F30` is actually the address to the first of the two bytes of memory where the `myNumber` integer value is stored. When the computer goes to `0x7F30` to read an integer value, it knows that it must read both the first and the second byte, since integers are two bytes on this specific computer.

The image below shows how the variable `myNumber = 17` is stored in memory.



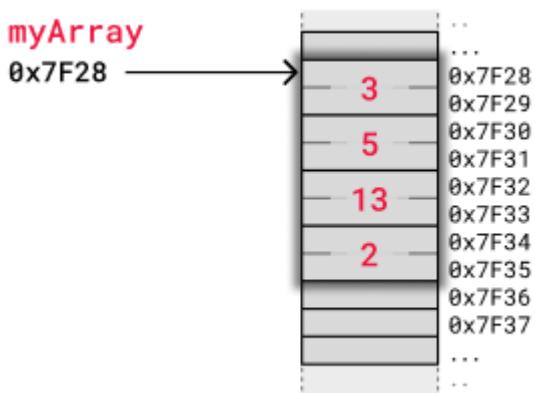
The example above shows how an integer value is stored on the simple, but popular, Arduino Uno microcontroller. This microcontroller has an 8 bit architecture with 16 bit address bus and uses two bytes for integers and two bytes for memory addresses. For comparison, personal computers and smart phones use 32 or 64 bits for integers and addresses, but the memory works basically in the same way.

## Arrays in Memory

To understand linked lists, it is useful to first know how arrays are stored in memory.

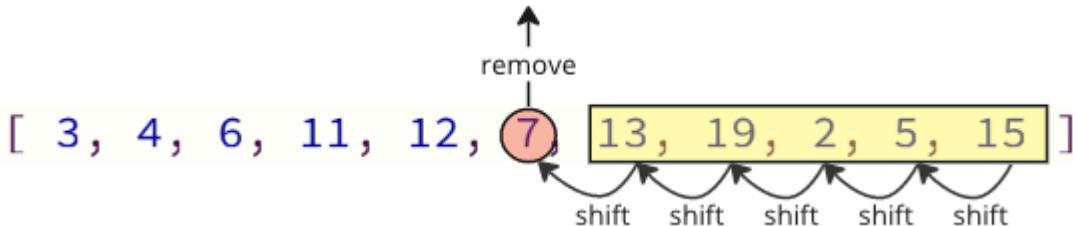
Elements in an array are stored contiguously in memory. That means that each element is stored right after the previous element.

The image below shows how an array of integers `myArray = [3, 5, 13, 2]` is stored in memory. We use a simple kind of memory here with two bytes for each integer, like in the previous example, just to get the idea.



The computer has only got the address of the first byte of `myArray`, so to access the 3rd element with code `myArray[2]` the computer starts at `0x7F28` and jumps over the two first integers. The computer knows that an integer is stored in two bytes, so it jumps  $2 \times 2$  bytes forward from `0x7F28` and reads value 13 starting at address `0x7F32`.

The image below shows how elements are shifted when an array element is removed.



Manipulating arrays is also something you must think about if you are programming in C, where you have to explicitly move other elements when inserting or removing an element. In C this does not happen in the background.

In C you also need to make sure that you have allocated enough space for the array to start with, so that you can add more elements later.

You can read more about arrays on [this previous DSA tutorial page](#).

## Linked Lists in Memory

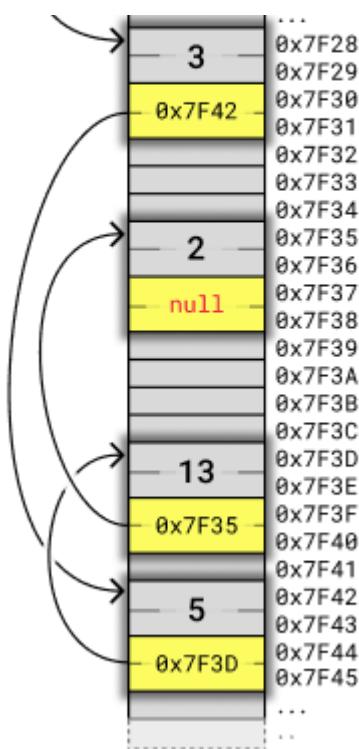
Instead of storing a collection of data as an array, we can create a linked list.

Linked lists are used in many scenarios, like dynamic data storage, stack and queue implementation or graph representation, to mention some of them.

A linked list consists of nodes with some sort of data, and at least one pointer, or link, to other nodes.

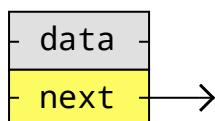
A big benefit with using linked lists is that nodes are stored wherever there is free space in memory, the nodes do not have to be stored contiguously right after each other like elements are stored in arrays. Another nice thing with linked lists is that when adding or removing nodes, the rest of the nodes in the list do not have to be shifted.

The image below shows how a linked list can be stored in memory. The linked list has four nodes with values 3, 5, 13 and 2, and each node has a pointer to the next node in the list.

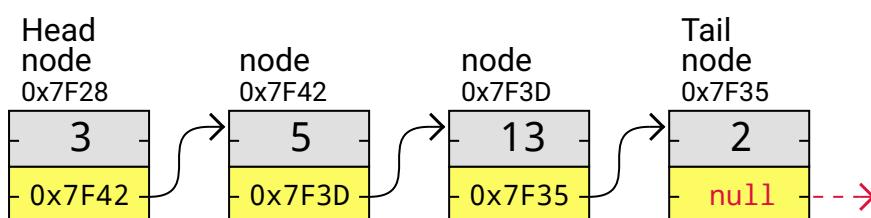


Each node takes up four bytes. Two bytes are used to store an integer value, and two bytes are used to store the address to the next node in the list. As mentioned before, how many bytes that are needed to store integers and addresses depend on the architecture of the computer. This example, like the previous array example, fits with a simple 8-bit microcontroller architecture.

To make it easier to see how the nodes relate to each other, we will display nodes in a linked list in a simpler way, less related to their memory location, like in the image below:



If we put the same four nodes from the previous example together using this new visualization, it looks like this:



As you can see, the first node in a linked list is called the "Head", and the last node is called the "Tail".

Something not so good with linked lists is that we cannot access a node directly like we can with an array by just writing `myArray[5]` for example. To get to node number 5 in a linked list, we must start with the first node called "head", use that node's pointer to get to the next node, and do so while keeping track of the number of nodes we have visited until we reach node number 5.

Learning about linked lists helps us to better understand concepts like memory allocation and pointers.

Linked lists are also important to understand before learning about more complex data structures such as trees and graphs, that can be implemented using linked lists.

## Memory in Modern Computers

So far on this page we have used the memory in an 8 bit microcontroller as an example to keep it simple and easier to understand.

Memory in modern computers work in the same way in principle as memory in an 8 bit microcontroller, but more memory is used to store integers, and more memory is used to store memory addresses.

The code below gives us the size of an integer and the size of a memory address on the server we are running these examples on.

## Example

Code written in C:

```
#include <stdio.h>

int main() {
    int myVal = 13;

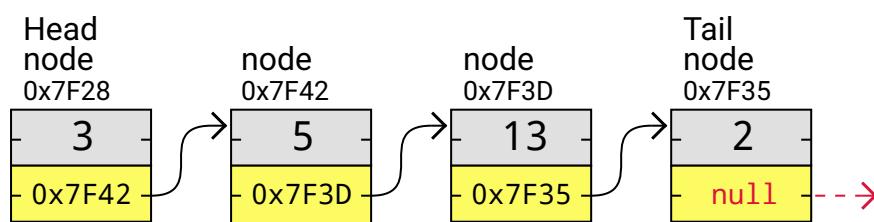
    printf("Value of integer 'myVal': %d\n", myVal);
    printf("Size of integer 'myVal': %lu bytes\n", sizeof(myVal)); // ←
    printf("Address to 'myVal': %p\n", &myVal);
    printf("Size of the address to 'myVal': %lu bytes\n", sizeof(&myVal))
```

[Run Example »](#)

The code example above only runs in C because Java and Python runs on an abstraction level above specific/direct memory allocation.

## Linked List Implementation in C

Let's implement this linked list from earlier:



Let's implement this linked list in C to see a concrete example of how linked lists are stored in memory.

In the code below, after including the libraries, we create a node struct which is like a class that represents what a node is: the node contains data and a pointer to the next node.

The `createNode()` function allocates memory for a new node, fills in the data part of the node with an integer given as an argument to the function, and returns the pointer (memory address) to the new node.

The `printList()` function is just for going through the linked list and printing each node's value.

Inside the `main()` function, four nodes are created, linked together, printed, and then the memory is freed. It is good practice to free memory after we are done using it to avoid memory leaks. Memory leak is when memory is not freed after use, gradually taking up more and more memory.

## Example

A basic linked list in C:

TLIN SASS VUE DSA GEN AI SCIPY AWS CYBERSECURITY DATA SC

```
typedef struct Node {
    int data;
    struct Node* next;
} Node;

Node* createNode(int data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    if (!newNode) {
        printf("Memory allocation failed!\n");
        exit(1);
    }
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

void printList(Node* node) {
    while (node) {
        printf("%d -> ", node->data);
        node = node->next;
    }
    printf("null\n");
}

int main() {
    Node* node1 = createNode(3);
    Node* node2 = createNode(5);
    Node* node3 = createNode(13);
    Node* node4 = createNode(2);

    node1->next = node2;
    node2->next = node3;
    node3->next = node4;

    printList(node1);

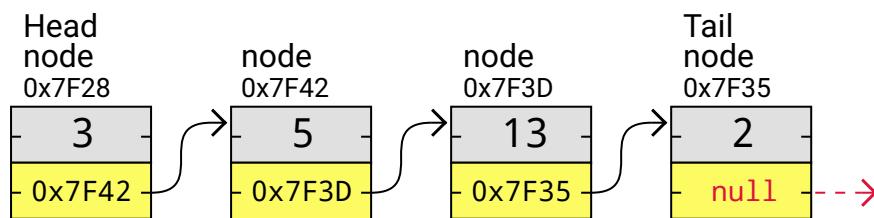
    // Free the memory
    free(node1);
    free(node2);
    free(node3);
    free(node4);
}
```

[Run Example »](#)

To print the linked list in the code above, the `printList()` function goes from one node to the next using the "next" pointers, and that is called "traversing" or "traversal" of the linked list. You will learn more about linked list traversal and other linked list operations on the [Linked Lists Operations page](#).

## Linked List Implementation in Python and Java

We will now implement this same linked list using Python and Java instead.



In the Python code below, the `Node` class represents what a node is: the node contains data and a link to the next node.

The `Node` class is used to create four nodes, the nodes are then linked together, and printed at the end.

As you can see, the Python code is a lot shorter than the C code, and perhaps better if you just want to understand the concept of linked lists, and not how linked lists are stored in memory.

The Java code is very similar to the Python code. Click the "Run Example" button below and choose the "Java" tab to see the Java code.

## Example

A basic linked list in Python:

```
class Node:  
    def __init__(self, data):
```

```
node2 = Node(5)
node3 = Node(13)
node4 = Node(2)

node1.next = node2
node2.next = node3
node3.next = node4

currentNode = node1
while currentNode:
    print(currentNode.data, end=" -> ")
    currentNode = currentNode.next
print("null")
```

[Run Example »](#)

## DSA Exercises

### Test Yourself With Exercises

#### Exercise:

What is a benefit of using Linked Lists?

A good thing about Linked Lists  
is that when inserting or  
removing a node, other elements  
do not have to be                   in memory.

# DSA Linked Lists Types

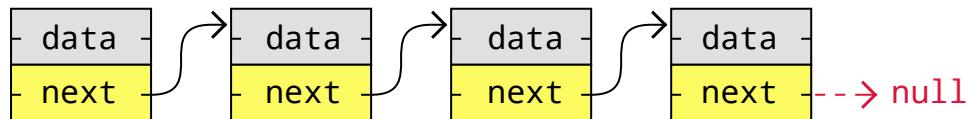
[« Previous](#)[Next »](#)

## Types of Linked Lists

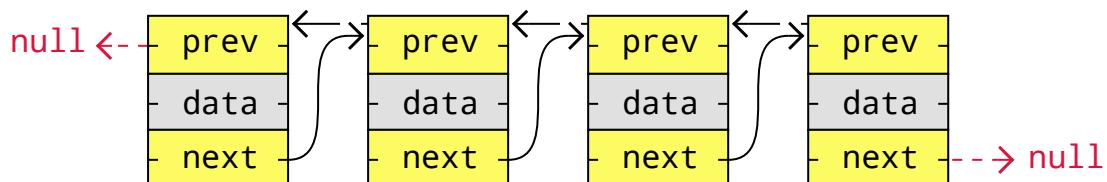
There are three basic forms of linked lists:

1. Singly linked lists
2. Doubly linked lists
3. Circular linked lists

A **singly linked list** is the simplest kind of linked lists. It takes up less space in memory because each node has only one address to the next node, like in the image below.



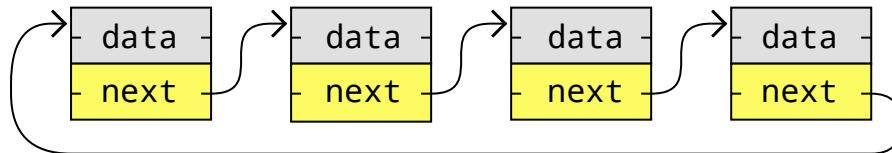
A **doubly linked list** has nodes with addresses to both the previous and the next node, like in the image below, and therefore takes up more memory. But doubly linked lists are good if you want to be able to move both up and down in the list.



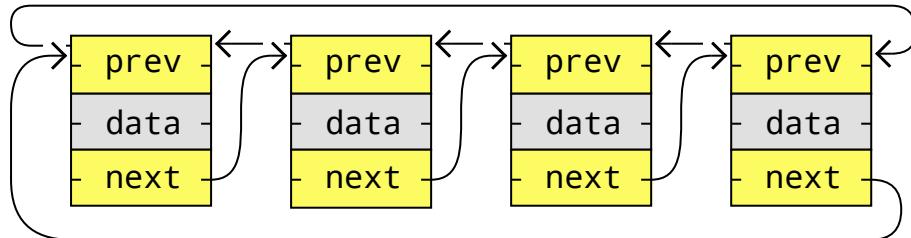
are `null`. But for circular linked lists, more complex code is needed to explicitly check for start and end nodes in certain applications.

Circular linked lists are good for lists you need to cycle through continuously.

The image below is an example of a singly circular linked list:



The image below is an example of a doubly circular linked list:



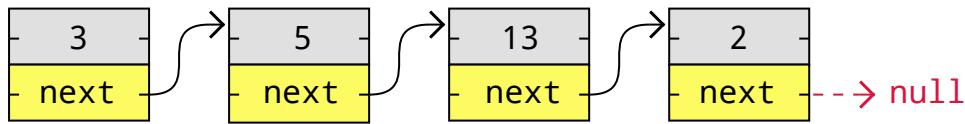
**Note:** What kind of linked list you need depends on the problem you are trying to solve.

## Linked List Implementations

Below are basic implementations of:

1. Singly linked list
2. Doubly linked list
3. Circular singly linked list
4. Circular doubly linked list

The next page will cover different operations that can be done on linked lists.



## Example

A basic singly linked list in Python:

(This is the same example as on the bottom of the previous page.)

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

node1 = Node(3)
node2 = Node(5)
node3 = Node(13)
node4 = Node(2)

node1.next = node2
node2.next = node3
node3.next = node4

currentNode = node1
while currentNode:
    print(currentNode.data, end=" -> ")
    currentNode = currentNode.next
print("null")
```

[Run Example »](#)

## 2. Doubly Linked List Implementation

Below is an implementation of this doubly linked list:

## Example

A basic doubly linked list in Python:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

node1 = Node(3)
node2 = Node(5)
node3 = Node(13)
node4 = Node(2)

node1.next = node2

node2.prev = node1
node2.next = node3

node3.prev = node2
node3.next = node4

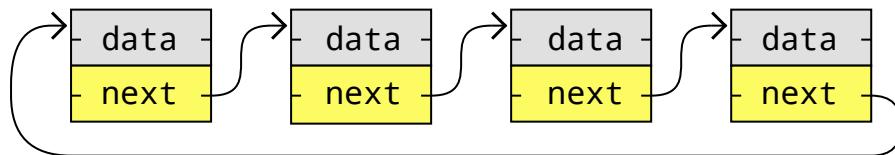
node4.prev = node3

print("\nTraversing forward:")
currentNode = node1
while currentNode:
    print(currentNode.data, end=" -> ")
    currentNode = currentNode.next
print("null")

print("\nTraversing backward:")
currentNode = node4
while currentNode:
    print(currentNode.data, end=" -> ")
    currentNode = currentNode.prev
print("null")
```

### 3. Circular Singly Linked List Implementation

Below is an implementation of this circular singly linked list:



#### Example

A basic circular singly linked list in Python:

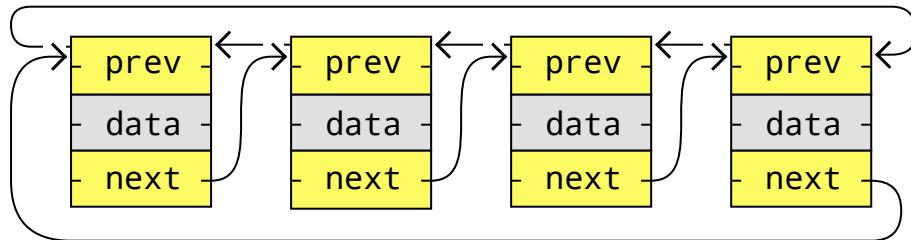
```
1  class Node:
2      def __init__(self, data):
3          self.data = data
4          self.next = None
5
6  node1 = Node(3)
7  node2 = Node(5)
8  node3 = Node(13)
9  node4 = Node(2)
10
11 node1.next = node2
12 node2.next = node3
13 node3.next = node4
14
15 currentNode = node1
16
17
18 print(currentNode.data, end=" -> ")
19 currentNode = currentNode.next
20
21 while currentNode != startNode:
22     print(currentNode.data, end=" -> ")
23     currentNode = currentNode.next
24
25
```

**Line 14:** This makes the singly list circular.

**Line 17:** This is how the program knows when to stop so that it only goes through the list one time.

## 4. Circular Doubly Linked List Implementation

Below is an implementation of this circular doubly linked list:



### Example

A basic circular doubly linked list in Python:

```
1  class Node:
2      def __init__(self, data):
3          self.data = data
4          self.next = None
5          self.prev = None
6
7      node1 = Node(3)
8      node2 = Node(5)
9      node3 = Node(13)
10     node4 = Node(2)
11
12     node1.next = node2
13
14     node2.prev = node1
15
16
```

```
TLIN SASS VUE DSA GEN AI SCIPY AWS CYBERSECURITY DATA SC  
20  
21 | node4.prev = node3  
| -----  
23 |  
24 | print("\nTraversing forward:")  
25 | currentNode = node1  
| -----  
27 | print(currentNode.data, end=" -> ")  
28 | currentNode = currentNode.next  
29 |  
30 | while currentNode != startNode:  
31 |     print(currentNode.data, end=" -> ")  
32 |     currentNode = currentNode.next  
33 | print("...")  
34 |  
35 | print("\nTraversing backward:")  
36 | currentNode = node4  
37 | startNode = node4  
38 | print(currentNode.data, end=" -> ")  
39 | currentNode = currentNode.prev  
40 |  
41 | while currentNode != startNode:  
42 |     print(currentNode.data, end=" -> ")  
43 |     currentNode = currentNode.prev  
44 | print("...")
```

[Run Example »](#)

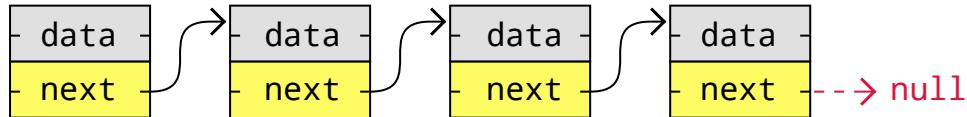
**Lines 13 and 22:** These links makes the doubly linked list circular.

**Lines 26:** This is how the program knows when to stop so that it only goes through the list one time.

## DSA Exercises

# Exercise:

Take a look at this singly Linked List:



How can we make this Linked List circular?

The list can be made circular by connecting the next pointer in the last node, to the node.

[Submit Answer »](#)

[Start the Exercise](#)

[!\[\]\(01508aa0c7d420981811331baf6678bc\_img.jpg\) Previous](#)

[Next !\[\]\(99dbe175fa5a0d5e993701c13a1b8079\_img.jpg\)](#)

**W3schools Pathfinder**  
Track your progress - it's free!

[Sign Up](#) [Log in](#)

# DSA Linked Lists Operations

[« Previous](#)[Next »](#)

## Linked List Operations

Basic things we can do with linked lists are:

1. Traversal
2. Remove a node
3. Insert a node
4. Sort

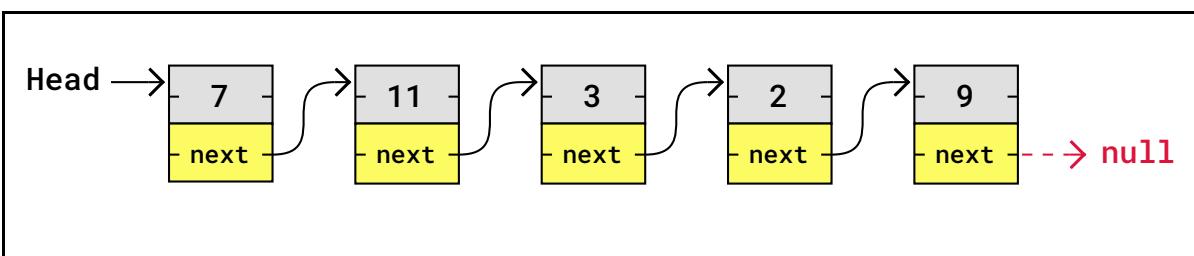
For simplicity, singly linked lists will be used to explain these operations below.

## Traversal of a Linked List

Traversing a linked list means to go through the linked list by following the links from one node to the next.

Traversal of linked lists is typically done to search for a specific node, and read or modify the node's content, remove the node, or insert a node right before or after that node.

To traverse a singly linked list, we start with the first node in the list, the head node, and follow that node's next link, and the next node's next link and so on, until the next address is null, like in the animation below:



the animation above.

## Example

Traversal of a singly linked list in Python:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

def traverseAndPrint(head):
    currentNode = head
    while currentNode:
        print(currentNode.data, end=" -> ")
        currentNode = currentNode.next
    print("null")

node1 = Node(7)
node2 = Node(11)
node3 = Node(3)
node4 = Node(2)
node5 = Node(9)

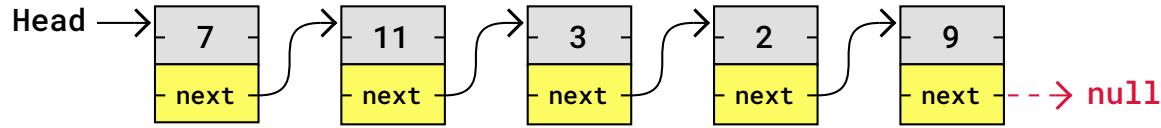
node1.next = node2
node2.next = node3
node3.next = node4
node4.next = node5

traverseAndPrint(node1)
```

[Run Example »](#)

## Find The Lowest Value in a Linked List

Let's find the lowest value in a singly linked list by traversing it and checking each value.



Lowest value:

**Find Lowest**

To find the lowest value we need to traverse the list like in the previous code. But in addition to traversing the list, we must also update the current lowest value when we find a node with a lower value.

In the code below, the algorithm to find the lowest value is moved into a function called `findLowestValue`.

## Example

Finding the lowest value in a singly linked list in Python:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

def findLowestValue(head):
    minValue = head.data
    currentNode = head.next
    while currentNode:
        if currentNode.data < minValue:
            minValue = currentNode.data
        currentNode = currentNode.next
    return minValue

node1 = Node(7)
node2 = Node(11)
node3 = Node(3)
node4 = Node(2)
node5 = Node(9)
```

```
node4.next = node5
```

```
print("The lowest value in the linked list is:", findLowestValue(node1))
```

The marked lines above is the core of the algorithm. The initial lowest value is set to be the value of the first node. Then, if a lower value is found, the lowest value variable is updated.

[Run Example »](#)

## Delete a Node in a Linked List

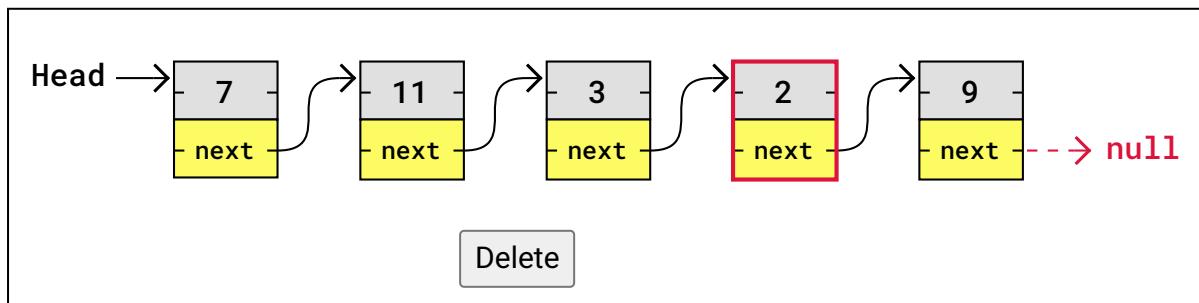
In this case we have the link (or pointer or address) to a node that we want to delete.

It is important to connect the nodes on each side of the node before deleting it, so that the linked list is not broken.

So before deleting the node, we need to get the next pointer from the previous node, and connect the previous node to the new next node before deleting the node in between.

In a singly linked list, like we have here, to get the next pointer from the previous node we actually need traverse the list from the start, because there is no way to go backwards from the node we want to delete.

The simulation below shows the node we want to delete, and how the list must be traversed first to connect the list properly before deleting the node without breaking the linked list.



Also, it is a good idea to first connect next pointer to the node after the node we want to delete, before we delete it. This is to avoid a 'dangling' pointer, a pointer that points to nothing, even if it is just for a brief moment.

In the code below, the algorithm to delete a node is moved into a function called

```
deleteSpecificNode .
```

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

def traverseAndPrint(head):
    currentNode = head
    while currentNode:
        print(currentNode.data, end=" -> ")
        currentNode = currentNode.next
    print("null")
```

```
node1 = Node(7)
node2 = Node(11)
node3 = Node(3)
node4 = Node(2)
node5 = Node(9)

node1.next = node2
node2.next = node3
node3.next = node4
node4.next = node5

print("Before deletion:")
traverseAndPrint(node1)
```

```
print("\nAfter deletion:")
traverseAndPrint(node1)
```

[Run Example »](#)

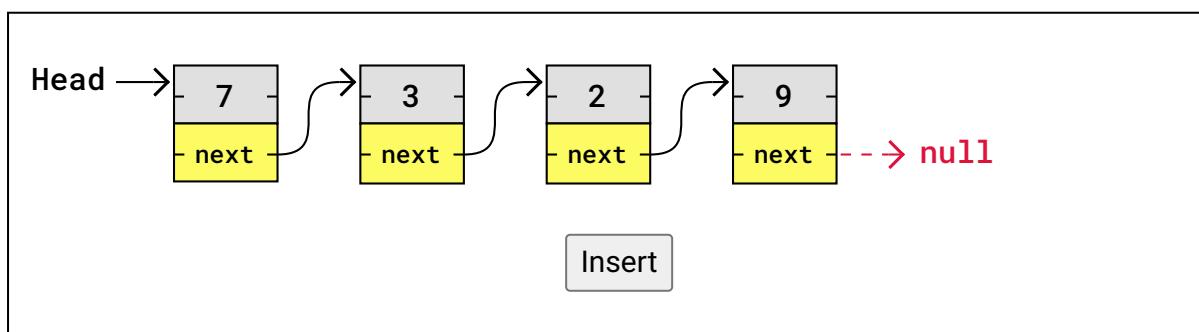
In the `deleteSpecificNode` function above, the return value is the new head of the linked list. So for example, if the node to be deleted is the first node, the new head returned will be the next node.

## Insert a Node in a Linked List

Inserting a node into a linked list is very similar to deleting a node, because in both cases we need to take care of the next pointers to make sure we do not break the linked list.

To insert a node in a linked list we first need to create the node, and then at the position where we insert it, we need to adjust the pointers so that the previous node points to the new node, and the new node points to the correct next node.

The simulation below shows how the links are adjusted when inserting a new node.



1. New node is created
2. Node 1 is linked to new node
3. New node is linked to next node

## Example

Inserting a node in a singly linked list in Python:

☰ TLIN SASS VUE DSA GEN AI SCIPY AWS CYBERSECURITY DATA SC

```
    self.next = None
```

```
def traverseAndPrint(head):
    currentNode = head
    while currentNode:
        print(currentNode.data, end=" -> ")
        currentNode = currentNode.next
    print("null")
```

```
node1 = Node(7)
```

```
node2 = Node(3)
```

```
node3 = Node(2)
```

```
node4 = Node(9)
```

```
node1.next = node2
```

```
node2.next = node3
```

```
node3.next = node4
```

```
print("Original list:")
```

```
traverseAndPrint(node1)
```

```
# Insert a new node with value 97 at position 2
```

```
newNode = Node(97)
```

```
node1 = insertNodeAtPosition(node1, newNode, 2)
```

```
print("\nAfter insertion:")
```

```
traverseAndPrint(node1)
```

In the `insertNodeAtPosition` function above, the return value is the new head of the linked list. So for example, if the node is inserted at the start of the linked list, the new head returned will be the new node.

## Other Linked Lists Operations

We have only covered three basic linked list operations above: traversal (or search), node deletion, and node insertion.

There are a lot of other operations that could be done with linked lists, like sorting for example.

Previously in the tutorial we have covered many sorting algorithms, and we could do many of these sorting algorithms on linked lists as well. Let's take selection sort for example. In selection sort we find the lowest value, remove it, and insert it at the beginning. We could do the same with a linked list as well, right? We have just seen how to search through a linked list, how to remove a node, and how to insert a node.

**Note:** We cannot sort linked lists with sorting algorithms like Counting Sort, Radix Sort or Quicksort because they use indexes to modify array elements directly based on their position.

## Linked Lists vs Arrays

These are some key linked list properties, compared to arrays:

- Linked lists are not allocated to a fixed size in memory like arrays are, so linked lists do not require to move the whole list into a larger memory space when the fixed memory space fills up, like arrays must.
- Linked list nodes are not laid out one right after the other in memory (contiguously), so linked list nodes do not have to be shifted up or down in memory when nodes are inserted or deleted.
- Linked list nodes require more memory to store one or more links to other nodes. Array elements do not require that much memory, because array elements do not contain links to other elements.
- Linked list operations are usually harder to program and require more lines than similar array operations, because programming languages have better built in support for arrays.

**Note:** When using arrays in programming languages like Java or Python, even though we do not need to write code to handle when an array fills up its memory space, and we do not have to shift elements up or down in memory when an element is removed or inserted, these things still happen in the background and can cause problems in time critical applications.

## Time Complexity of Linked Lists Operations

Here we discuss time complexity of linked list operations, and compare these with the time complexity of the array algorithms that we have discussed previously in this tutorial.

Remember that time complexity just says something about the approximate number of operations needed by the algorithm based on a large set of data  $n$ , and does not tell us the exact time a specific implementation of an algorithm takes.

This means that even though linear search is said to have the same time complexity for arrays as for linked list:  $O(n)$ , it does not mean they take the same amount of time. The exact time it takes for an algorithm to run depends on programming language, computer hardware, differences in time needed for operations on arrays vs linked lists, and many other things as well.

Linear search for linked lists works the same as for arrays. A list of unsorted values are traversed from the head node until the node with the specific value is found. Time complexity is  $O(n)$ .

Binary search is not possible for linked lists because the algorithm is based on jumping directly to different array elements, and that is not possible with linked lists.

Sorting algorithms have the same time complexities as for arrays, and these are explained earlier in this tutorial. But remember, sorting algorithms that are based on directly accessing an array element based on an index, do not work on linked lists.

## DSA Exercises

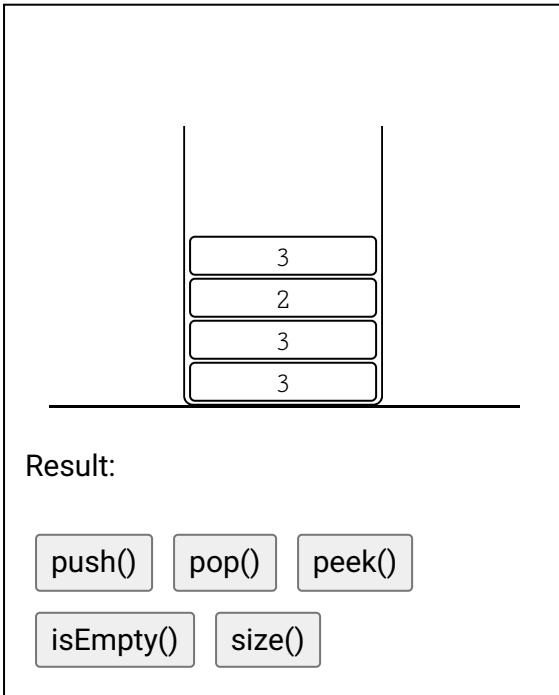
### Test Yourself With Exercises

# DSA Stacks

[« Previous](#)[Next »](#)

## Stacks

A stack is a data structure that can hold many elements.



Think of a stack like a pile of pancakes.

In a pile of pancakes, the pancakes are both added and removed from the top. So when removing a pancake, it will always be the last pancake you added. This way of organizing elements is called LIFO: Last In First Out.

Basic operations we can do on a stack are:

- **Push:** Adds a new element on the stack.

Experiment with these basic operations in the stack animation above.

Stacks can be implemented by using arrays or linked lists.

Stacks can be used to implement undo mechanisms, to revert to previous states, to create algorithms for depth-first search in graphs, or for backtracking.

Stacks are often mentioned together with Queues, which is a similar data structure described on the next page.

## Stack Implementation using Arrays

To better understand the benefits with using arrays or linked lists to implement stacks, you should check out [this page](#) that explains how arrays and linked lists are stored in memory.

This is how it looks like when we use an array as a stack:

[ 2, 2, 1, 3 ]

Result:

push() pop() peek()

isEmpty() size()

Reasons to implement stacks using arrays:

- **Memory Efficient:** Array elements do not hold the next elements address like linked list nodes do.
- **Easier to implement and understand:** Using arrays to implement stacks require less code than using linked lists, and for this reason it is typically easier to understand as well.

A reason for **not** using arrays to implement stacks:

- **Fixed size:** An array occupies a fixed part of the memory. This means that it could take up more memory than needed, or if the array fills up, it cannot hold more elements.

**Note:** When using arrays in Python for this tutorial, we are really using the Python 'list' data type, but for the scope of this tutorial the 'list' data type can be used in the same way as an array. Learn

Since Python lists has good support for functionality needed to implement stacks, we start with creating a stack and do stack operations with just a few lines like this:

## Example

Python:

```
stack = []

# Push
stack.append('A')
stack.append('B')
stack.append('C')
print("Stack: ", stack)

# Pop
element = stack.pop()
print("Pop: ", element)

# Peek
topElement = stack[-1]
print("Peek: ", topElement)

# isEmpty
isEmpty = not bool(stack)
print("isEmpty: ", isEmpty)

# Size
print("Size: ", len(stack))
```

[Run Example »](#)

But to explicitly create a data structure for stacks, with basic operations, we should create a stack class instead. This way of creating stacks in Python is also more similar to how stacks can be created in other programming languages like C and Java.

## Example

≡ TLIN SASS VUE DSA GEN AI SCIPY AWS CYBERSECURITY DATA SC

```
self.stack = []

def push(self, element):
    self.stack.append(element)

def pop(self):
    if self.isEmpty():
        return "Stack is empty"
    return self.stack.pop()

def peek(self):
    if self.isEmpty():
        return "Stack is empty"
    return self.stack[-1]

def isEmpty(self):
    return len(self.stack) == 0

def size(self):
    return len(self.stack)

# Create a stack
myStack = Stack()

myStack.push('A')
myStack.push('B')
myStack.push('C')
print("Stack: ", myStack.stack)

print("Pop: ", myStack.pop())

print("Peek: ", myStack.peek())

print("isEmpty: ", myStack.isEmpty())

print("Size: ", myStack.size())
```

[Run Example »](#)

- **Dynamic size:** The stack can grow and shrink dynamically, unlike with arrays.

Reasons for **not** using linked lists to implement stacks:

- **Extra memory:** Each stack element must contain the address to the next element (the next linked list node).
- **Readability:** The code might be harder to read and write for some because it is longer and more complex.

This is how a stack can be implemented using a linked list.

## Example

Python:

```
class Node:  
    def __init__(self, value):  
        self.value = value  
        self.next = None  
  
class Stack:  
    def __init__(self):  
        self.head = None  
        self.size = 0  
  
    def push(self, value):  
        new_node = Node(value)  
        if self.head:  
            new_node.next = self.head  
        self.head = new_node  
        self.size += 1  
  
    def pop(self):  
        if self.isEmpty():  
            return "Stack is empty"  
        popped_node = self.head  
        self.head = self.head.next  
        self.size -= 1  
        return popped_node.value
```

```
def isEmpty(self):
    return self.size == 0

def stackSize(self):
    return self.size

myStack = Stack()
myStack.push('A')
myStack.push('B')
myStack.push('C')

print("Pop: ", myStack.pop())
print("Peek: ", myStack.peek())
print("isEmpty: ", myStack.isEmpty())
print("Size: ", myStack.stackSize())
```

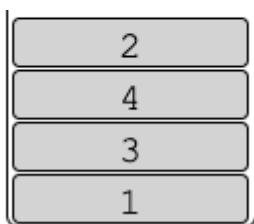
[Run Example »](#)

## DSA Exercises

Test Yourself With Exercises

### Exercise:

The image below represents a "Stack" data structure.



Running the `peek()` method on the Stack above, what is returned?

[Submit Answer »](#)

[Start the Exercise](#)

[« Previous](#)

[Next »](#)

**W3schools Pathfinder**  
Track your progress - it's free!

[Sign Up](#)   [Log in](#)

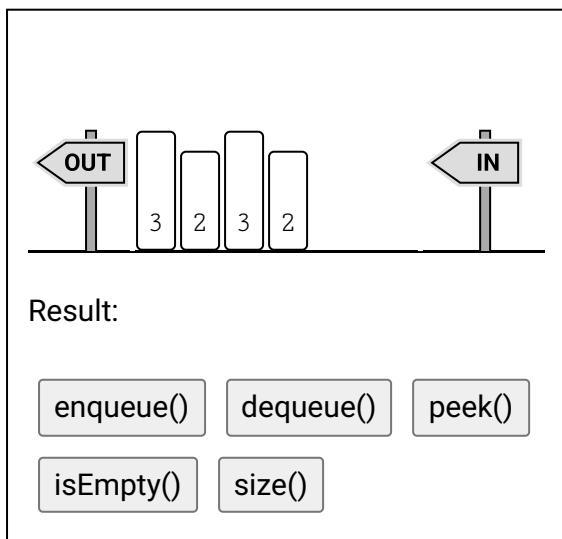
ADVERTISEMENT

# DSA Queues

[« Previous](#)[Next »](#)

## Queues

A queue is a data structure that can hold many elements.



Think of a queue as people standing in line in a supermarket.

The first person to stand in line is also the first who can pay and leave the supermarket. This way of organizing elements is called FIFO: First In First Out.

Basic operations we can do on a queue are:

- **Enqueue:** Adds a new element to the queue.
- **Dequeue:** Removes and returns the first (front) element from the queue.
- **Peek:** Returns the first element in the queue.
- **isEmpty:** Checks if the queue is empty.
- **Size:** Finds the number of elements in the queue.

Queues can be used to implement job scheduling for an office printer, order processing for e-tickets, or to create algorithms for breadth-first search in graphs.

Queues are often mentioned together with Stacks, which is a similar data structure described on the [previous page](#).

# Queue Implementation using Arrays

To better understand the benefits with using arrays or linked lists to implement queues, you should check out [this page](#) that explains how arrays and linked lists are stored in memory.

This is how it looks like when we use an array as a queue:

[ 1, 4, 2, 1 ]

Result:

enqueue() dequeue() peek()

isEmpty() size()

Reasons to implement queues using arrays:

- **Memory Efficient:** Array elements do not hold the next elements address like linked list nodes do.
  - **Easier to implement and understand:** Using arrays to implement queues require less code than using linked lists, and for this reason it is typically easier to understand as well.

Reasons for **not** using arrays to implement queues:

- **Fixed size:** An array occupies a fixed part of the memory. This means that it could take up more memory than needed, or if the array fills up, it cannot hold more elements. And resizing an array can be costly.
  - **Shifting cost:** Dequeue causes the first element in a queue to be removed, and the other elements must be shifted to take the removed elements' place. This is inefficient and can cause problems, especially if the queue is long.
  - **Alternatives:** Some programming languages have built-in data structures optimized for queue operations that are better than using arrays.

Since Python lists has good support for functionality needed to implement queues, we start with creating a queue and do queue operations with just a few lines:

## Example

Python:

```
queue = []

# Enqueue
queue.append('A')
queue.append('B')
queue.append('C')
print("Queue: ", queue)

# Dequeue
element = queue.pop(0)
print("Dequeue: ", element)

# Peek
frontElement = queue[0]
print("Peek: ", frontElement)

# isEmpty
isEmpty = not bool(queue)
print("isEmpty: ", isEmpty)

# Size
print("Size: ", len(queue))
```

[Run Example »](#)

But to explicitly create a data structure for queues, with basic operations, we should create a queue class instead. This way of creating queues in Python is also more similar to how queues can be created in other programming languages like C and Java.

```
class Queue:  
    def __init__(self):  
        self.queue = []  
  
    def enqueue(self, element):  
        self.queue.append(element)  
  
    def dequeue(self):  
        if self.isEmpty():  
            return "Queue is empty"  
        return self.queue.pop(0)  
  
    def peek(self):  
        if self.isEmpty():  
            return "Queue is empty"  
        return self.queue[0]  
  
    def isEmpty(self):  
        return len(self.queue) == 0  
  
    def size(self):  
        return len(self.queue)  
  
# Create a queue  
myQueue = Queue()  
  
myQueue.enqueue('A')  
myQueue.enqueue('B')  
myQueue.enqueue('C')  
print("Queue: ", myQueue.queue)  
  
print("Dequeue: ", myQueue.dequeue())  
  
print("Peek: ", myQueue.peek())  
  
print("isEmpty: ", myQueue.isEmpty())  
  
print("Size: ", myQueue.size())
```

# Queue Implementation using Linked Lists

Reasons for using linked lists to implement queues:

- **Dynamic size:** The queue can grow and shrink dynamically, unlike with arrays.
- **No shifting:** The front element of the queue can be removed (enqueue) without having to shift other elements in the memory.

Reasons for **not** using linked lists to implement queues:

- **Extra memory:** Each queue element must contain the address to the next element (the next linked list node).
- **Readability:** The code might be harder to read and write for some because it is longer and more complex.

This is how a queue can be implemented using a linked list.

## Example

Python:

```
class Node:  
    def __init__(self, data):  
        self.data = data  
        self.next = None  
  
class Queue:  
    def __init__(self):  
        self.front = None  
        self.rear = None  
        self.length = 0  
  
    def enqueue(self, element):  
        new_node = Node(element)  
        if self.rear is None:  
            self.front = self.rear = new_node  
            self.length += 1  
            return  
        self.rear.next = new_node  
        self.rear = new_node
```

```
        return "Queue is empty"
    temp = self.front
    self.front = temp.next
    self.length -= 1
    if self.front is None:
        self.rear = None
    return temp.data

def peek(self):
    if self.isEmpty():
        return "Queue is empty"
    return self.front.data

def isEmpty(self):
    return self.length == 0

def size(self):
    return self.length

def printQueue(self):
    temp = self.front
    while temp:
        print(temp.data, end=" ")
        temp = temp.next
    print()

# Create a queue
myQueue = Queue()

myQueue.enqueue('A')
myQueue.enqueue('B')
myQueue.enqueue('C')
print("Queue: ", end="")
myQueue.printQueue()

print("Dequeue: ", myQueue.dequeue())

print("Peek: ", myQueue.peek())

print("isEmpty: ", myQueue.isEmpty())
```

# DSA Hash Tables

[« Previous](#)[Next »](#)

## Hash Table

A Hash Table is a data structure designed to be fast to work with.

The reason Hash Tables are sometimes preferred instead of arrays or linked lists is because searching for, adding, and deleting data can be done really quickly, even for large amounts of data.

In a [Linked List](#), finding a person "Bob" takes time because we would have to go from one node to the next, checking each node, until the node with "Bob" is found.

And finding "Bob" in an [Array](#) could be fast if we knew the index, but when we only know the name "Bob", we need to compare each element (like with Linked Lists), and that takes time.

With a Hash Table however, finding "Bob" is done really fast because there is a way to go directly to where "Bob" is stored, using something called a hash function.

## Building A Hash Table from Scratch

To get the idea of what a Hash Table is, let's try to build one from scratch, to store unique first names inside it.

We will build the Hash Set in 5 steps:

1. Starting with an array.
2. Storing names using a hash function.
3. Looking up an element using a hash function.
4. Handling collisions.
5. The basic Hash Set code example and simulation.

```
my_array = ['Pete', 'Jones', 'Lisa', 'Bob', 'Siri']
```

To find "Bob" in this array, we need to compare each name, element by element, until we find "Bob".

If the array was sorted alphabetically, we could use Binary Search to find a name quickly, but inserting or deleting names in the array would mean a big operation of shifting elements in memory.

To make interacting with the list of names really fast, let's use a Hash Table for this instead, or a Hash Set, which is a simplified version of a Hash Table.

To keep it simple, let's assume there is at most 10 names in the list, so the array must be a fixed size of 10 elements. When talking about Hash Tables, each of these elements is called a **bucket**.

```
my_hash_set = [None, None, None, None, None, None, None, None, None, None]
```

## Step 2: Storing names using a hash function

Now comes the special way we interact with the Hash Set we are making.

We want to store a name directly into its right place in the array, and this is where the **hash function** comes in.

A hash function can be made in many ways, it is up to the creator of the Hash Table. A common way is to find a way to convert the value into a number that equals one of the Hash Set's index numbers, in this case a number from 0 to 9. In our example we will use the Unicode number of each character, summarize them and do a modulo 10 operation to get index numbers 0-9.

## Example

```
def hash_function(value):  
    sum_of_chars = 0
```

```
print("'Bob' has hash code:",hash_function('Bob'))
```

[Run Example »](#)

The character "B" has Unicode code point 66, "o" has 111, and "b" has 98. Adding those together we get 275. Modulo 10 of 275 is 5, so "Bob" should be stored as an array element at index 5.

The number returned by the hash function is called the **hash code**.

**Unicode number:** Everything in our computers are stored as numbers, and the Unicode code point is a unique number that exist for every character. For example, the character  has Unicode number (also called Unicode code point)  . Just try it in the simulation below. See [this page](#) for more information about how characters are represented as numbers.

**Modulo:** A mathematical operation, written as  in most programming languages (or *mod* in mathematics). A modulo operation divides a number with another number, and gives us the resulting remainder. So for example,  will give us the remainder  . (Dividing 7 apples between 3 people, means that each person gets 2 apples, with 1 apple to spare.)

After storing "Bob" where the hash code tells us (index 5), our array now looks like this:

```
my_hash_set = [None, None, None, None, None, 'Bob', None, None, None, None]
```

We can use the hash function to find out where to store the other names "Pete", "Jones", "Lisa", and "Siri" as well.

After using the hash function to store those names in the correct position, our array looks like this:

```
my_hash_set = [None, 'Jones', None, 'Lisa', None, 'Bob', None, 'Siri', 'Pete', None]
```

element by element anymore to find out if "Pete" is in there, we can just use the hash function to go straight to the right element!

To find out if "Pete" is stored in the array, we give the name "Pete" to our hash function, we get back hash code 9, we go directly to the element at index 9, and there he is. We found "Pete" without checking any other elements.

## Example

```
my_hash_set = [None, 'Jones', None, 'Lisa', None, 'Bob', None, 'Siri', 'Pete', None]

def hash_function(value):
    sum_of_chars = 0
    for char in value:
        sum_of_chars += ord(char)

    return sum_of_chars % 10

def contains(name):
    index = hash_function(name)
    return my_hash_set[index] == name

print("'Pete' is in the Hash Set:", contains('Pete'))
```

[Run Example »](#)

When deleting a name from our Hash Set, we can also use the hash function to go straight to where the name is, and set that element value to `None`.

## Step 4: Handling collisions

Let's also add "Stuart" to our Hash Set.

We give "Stuart" to our hash function, and we get the hash code 3, meaning "Stuart" should be stored at index 3.

collision problem in this way is called chaining. We can give room for more elements in the same bucket by implementing each bucket as a linked list, or as an array.

After implementing each bucket as an array, to give room for potentially more than one name in each bucket, "Stuart" can also be stored at index 3, and our Hash Set now looks like this:

```
my_hash_set = [
    [None],
    ['Jones'],
    [None],
    ['Lisa', 'Stuart'],
    [None],
    ['Bob'],
    [None],
    ['Siri'],
    ['Pete'],
    [None]
]
```

Searching for "Stuart" in our Hash Set now means that using the hash function we end up directly in bucket 3, but then we must first check "Lisa" in that bucket, before we find "Stuart" as the second element in bucket 3.

## Step 5: Hash Set code example and simulation

To complete our very basic Hash Set code, let's have functions for adding and searching for names in the Hash Set, which is now a two dimensional array.

Run the code example below, and try it with different values to get a better understanding of how a Hash Set works.

## Example

```
1 my_hash_set = [
2     [None],
3     ['Jones'],
4
```

☰ TLIN SASS VUE DSA GEN AI SCIPY AWS CYBERSECURITY DATA SC

```
 8     [None],  
 9     ['Siri'],  
10    ['Pete'],  
11    [None]  
12 ]  
13  
14 def hash_function(value):  
15     return sum(ord(char) for char in value) % 10  
16  
17 def add(value):  
18     index = hash_function(value)  
19     bucket = my_hash_set[index]  
20     if value not in bucket:  
21         bucket.append(value)  
22  
23 def contains(value):  
24     index = hash_function(value)  
25     bucket = my_hash_set[index]  
26     return value in bucket  
27  
28 add('Stuart')  
29  
30 print(my_hash_set)  
31 print('Contains Stuart:',contains('Stuart'))
```

[Run Example »](#)

The next two pages show better and more detailed implementations of Hash Sets and Hash Tables.

Try the Hash Set simulation below to get a better ide of how a Hash Set works in principle.

### Hash Set

```
0 : Thomas Jens  
1 :  
2 : Peter
```

TLIN    SASS    VUE    DSA    GEN AI    SCIPY    AWS    CYBERSECURITY    DATA SC

6 :

7 :

8 : Michaela

9 :

---

**Hash Code**

$0 \% 10 = 0$

---

Try interacting with the Hash Set

Write a name...

contains()add()remove()size()

## Uses of Hash Tables

Hash Tables are great for:

- Checking if something is in a collection (like finding a book in a library).
- Storing unique items and quickly finding them (like storing phone numbers).
- Connecting values to keys (like linking names to phone numbers).

The most important reason why Hash Tables are great for these things is that Hash Tables are very fast compared Arrays and Linked Lists, especially for large sets. Arrays and Linked Lists have time complexity  $O(n)$  for search and delete, while Hash Tables have just  $O(1)$  on average! Read more about time complexity [here](#).

## Hash Set vs. Hash Map

A Hash Table can be a Hash Set or a Hash Map. The next two pages describe these data structures in more detail.

Here's how Hash Sets and Hash Maps are different and similar:

a value connected it.

Use case	Checking if an element is in the set, like checking if a name is on a guest list.	Finding information based on a key, like looking up who owns a certain telephone number.
<i>Is it fast to search, add and delete elements?</i>	Yes, average $O(1)$ .	Yes, average $O(1)$ .
<i>Is there a hash function that takes the key, generates a hash code, and that is the bucket where the element is stored?</i>	Yes	Yes

## Hash Tables Summarized

Hash Table elements are stored in storage containers called **buckets**.

Every Hash Table element has a part that is unique that is called the **key**.

A **hash function** takes the key of an element to generate a **hash code**.

The hash code says what bucket the element belongs to, so now we can go directly to that Hash Table element: to modify it, or to delete it, or just to check if it exists. Specific hash functions are explained in detail on the next two pages.

A **collision** happens when two Hash Table elements have the same hash code, because that means they belong to the same **bucket**. A collision can be solved in two ways.

**Chaining** is the way collisions are solved in this tutorial, by using arrays or linked lists to allow more than one element in the same bucket.

**Open Addressing** is another way to solve collisions. With open addressing, if we want to store an element but there is already an element in that bucket, the element is stored in the next available bucket. This can be done in many different ways, but we will not explain open addressing any further here.

## Conclusion

[Tutorials ▾](#)[Exercises ▾](#)[Services ▾](#)[Sign Up](#)[Log in](#)[☰](#) [TLIN](#) [SASS](#) [VUE](#) [DSA](#) [GEN AI](#) [SCIPY](#) [AWS](#) [CYBERSECURITY](#) [DATA SC](#)

is there, or to find detailed information about it.

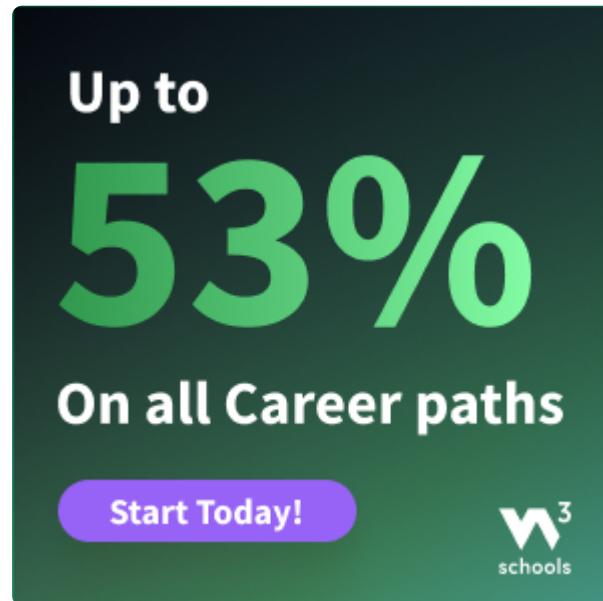
[\*\*« Previous\*\*](#)[\*\*Next »\*\*](#)

### W3schools Pathfinder

Track your progress - it's free!

[Sign Up](#) [Log in](#)

ADVERTISEMENT



### COLOR PICKER



# DSA Hash Sets

[« Previous](#)[Next »](#)

## Hash Sets

A Hash Set is a form of Hash Table data structure that usually holds a large number of elements.

Using a Hash Set we can search, add, and remove elements really fast.

Hash Sets are used for lookup, to check if an element is part of a set.

### Hash Set

0 : Thomas Jens

1 :

2 : Peter

3 : Lisa

4 : Charlotte

5 : Adele Bob

6 :

7 :

8 : Michaela

9 :

### Hash Code

$0 \% 10 = 0$

[contains\(\)](#) [add\(\)](#) [remove\(\)](#) [size\(\)](#)

A Hash Set stores unique elements in buckets according to the element's hash code.

- **Hash code:** A number generated from an element's unique value (key), to determine what bucket that Hash Set element belongs to.
- **Unique elements:** A Hash Set cannot have more than one element with the same value.
- **Bucket:** A Hash Set consists of many such buckets, or containers, to store elements. If two elements have the same hash code, they belong to the same bucket. The buckets are therefore often implemented as arrays or linked lists, because a bucket needs to be able to hold more than one element.

## Finding The Hash Code

A hash code is generated by a **hash function**.

The hash function in the animation above takes the name written in the input, and sums up the Unicode code points for every character in that name.

After that, the hash function does a modulo 10 operation (`% 10`) on the sum of characters to get the hash code as a number from 0 to 9.

This means that a name is put into one of ten possible buckets in the Hash Set, according to the hash code of that name. The same hash code is generated and used when we want to search for or remove a name from the Hash Set.

The Hash Code gives us instant access as long as there is just one name in the corresponding bucket.

**Unicode code point:** Everything in our computers are stored as numbers, and the Unicode code point is a unique number that exist for every character. For example, the character `A` has Unicode code point `65`. Just try it in the simulation above. See [this page](#) for more information about how characters are represented as numbers.

**Modulo:** A mathematical operation, written as `%` in most programming languages (or `mod` in mathematics). A modulo operation divides a number with another number, and gives us the

# Direct Access in Hash Sets

Searching for `Peter` in the Hash Set above, means that the hash code `2` is generated (`512 % 10`), and that directs us right to the bucket `Peter` is in. If that is the only name in that bucket, we will find `Peter` right away.

In cases like this we say that the Hash Set has constant time  $O(1)$  for searching, adding, and removing elements, which is really fast.

But, if we search for `Jens`, we need to search through the other names in that bucket before we find `Jens`. In a worst case scenario, all names end up in the same bucket, and the name we are searching for is the last one. In such a worst case scenario the Hash Set has time complexity  $O(n)$ , which is the same time complexity as arrays and linked lists.

To keep Hash Sets fast, it is therefore important to have a hash function that will distribute the elements evenly between the buckets, and to have around as many buckets as Hash Set elements.

Having a lot more buckets than Hash Set elements is a waste of memory, and having a lot less buckets than Hash Set elements is a waste of time.

## Hash Set Implementation

Hash Sets in Python are typically done by using Python's own `set` data type, but to get a better understanding of how Hash Sets work we will not use that here.

To implement a Hash Set in Python we create a class `SimpleHashSet`.

Inside the `SimpleHashSet` class we have a method `__init__` to initialize the Hash Set, a method `hash_function` for the hash function, and methods for the basic Hash Set operations: `add`, `contains`, and `remove`.

We also create a method `print_set` to better see how the Hash Set looks like.

## Example

```
TLIN SASS VUE DSA GEN AI SCIPY AWS CYBERSECURITY DATA SC
4         self.buckets = [[ ] for _ in range(size)] # A list of buckets
5
6     def hash_function(self, value):
7         # Simple hash function: sum of character codes modulo the size
8         return sum(ord(char) for char in value) % self.size
9
10    def add(self, value):
11        # Add a value if it's not already present
12        index = self.hash_function(value)
13        bucket = self.buckets[index]
14        if value not in bucket:
15            bucket.append(value)
16
17    def contains(self, value):
18        # Check if a value exists in the set
19        index = self.hash_function(value)
20        bucket = self.buckets[index]
21        return value in bucket
22
23    def remove(self, value):
24        # Remove a value
25        index = self.hash_function(value)
26        bucket = self.buckets[index]
27        if value in bucket:
28            bucket.remove(value)
29
30    def print_set(self):
31        # Print all elements in the hash set
32        print("Hash Set Contents:")
33        for index, bucket in enumerate(self.buckets):
34            print(f"Bucket {index}: {bucket}")
```

Using the `SimpleHashSet` class we can create the same Hash Set as in the top of this page:

## Example

```
42 | hash_set.add("Peter")
43 | hash_set.add("Lisa")
44 | hash_set.add("Adele")
45 | hash_set.add("Michaela")
46 | hash_set.add("Bob")
47 |
48 | hash_set.print_set()
49 |

print("\n'Peter' is in the set:",hash_set.contains('Peter'))
print("Removing 'Peter'")
hash_set.remove('Peter')
print("'Peter' is in the set:",hash_set.contains('Peter'))
print("'Adele' has hash code:",hash_set.hash_function('Adele'))
```

[Run Example »](#)[« Previous](#)[Next »](#)

**W3schools Pathfinder**  
Track your progress - it's free!

[Sign Up](#) [Log in](#)

ADVERTISEMENT

# DSA Hash Maps

[« Previous](#)[Next »](#)

## Hash Maps

A Hash Map is a form of Hash Table data structure that usually holds a large number of entries.

Using a Hash Map we can search, add, modify, and remove entries really fast.

Hash Maps are used to find detailed information about something.

In the simulation below, people are stored in a Hash Map. A person can be looked up using a person's unique social security number (the Hash Map key), and then we can see that person's name (the Hash Map value).

### Hash Map

0 :	<table border="1"><tr><td>123-4569</td><td>Jens</td></tr></table>	123-4569	Jens		
123-4569	Jens				
1 :					
2 :	<table border="1"><tr><td>123-4570</td><td>Peter</td></tr></table>	123-4570	Peter		
123-4570	Peter				
3 :	<table border="1"><tr><td>123-4571</td><td>Lisa</td></tr></table>	123-4571	Lisa		
123-4571	Lisa				
4 :					
5 :	<table border="1"><tr><td>123-4672</td><td>Adele</td><td>123-4573</td><td>Michaela</td></tr></table>	123-4672	Adele	123-4573	Michaela
123-4672	Adele	123-4573	Michaela		
6 :					
7 :					
8 :	<table border="1"><tr><td>123-4567</td><td>Charlotte</td><td>123-6574</td><td>Bob</td></tr></table>	123-4567	Charlotte	123-6574	Bob
123-4567	Charlotte	123-6574	Bob		
9 :	<table border="1"><tr><td>123-4568</td><td>Thomas</td></tr></table>	123-4568	Thomas		
123-4568	Thomas				

Try interacting with the Hash Map

XXX	- XXXX	Name
put()	remove()	get()
size()		

**Note:** The Hash Map would be more useful if more information about each person was attached to the corresponding social security number, like last name, birth date, and address, and maybe other things as well. But the Hash Map simulation above is made to be as simple as possible.

It is easier to understand how Hash Maps work if you first have a look at the two previous pages about [Hash Tables](#) and [Hash Sets](#). It is also important to understand the meaning of the words below.

- **Entry:** Consists of a key and a value, forming a key-value pair.
- **Key:** Unique for each entry in the Hash Map. Used to generate a hash code determining the entry's bucket in the Hash Map. This ensures that every entry can be efficiently located.
- **Hash Code:** A number generated from an entry's key, to determine what bucket that Hash Map entry belongs to.
- **Bucket:** A Hash Map consists of many such buckets, or containers, to store entries.
- **Value:** Can be nearly any kind of information, like name, birth date, and address of a person. The value can be many different kinds of information combined.

## Finding The Hash Code

A hash code is generated by a **hash function**.

The hash function in the simulation above takes the numbers in the social security number (not the dash), add them together, and does a modulo 10 operation ( $\% 10$ ) on the sum of characters to get the hash code as a number from 0 to 9.

This means that a person is stored in one of ten possible buckets in the Hash Map, according to the hash code of that person's social security number. The same hash code is generated and used when we want to search for or remove a person from the Hash Map.

numbers together gives us a sum  $28$ , and modulo 10 of that is  $8$ . That is why she belongs to bucket  $8$ .

**Modulo:** A mathematical operation, written as  $\%$  in most programming languages (or *mod* in mathematics). A modulo operation divides a number with another number, and gives us the resulting remainder. So for example,  $7 \% 3$  will give us the remainder  $1$ . (Dividing 7 apples between 3 people, means that each person gets 2 apples, with 1 apple to spare.)

## Direct Access in Hash Maps

Searching for `Charlotte` in the Hash Map, we must use the social security number `123 - 4567` (the Hash Map key), which generates the hash code  $8$ , as explained above.

This means we can go straight to bucket  $8$  to get her name (the Hash Map value), without searching through other entries in the Hash Map.

In cases like this we say that the Hash Map has constant time  $O(1)$  for searching, adding, and removing entries, which is really fast compared to using an array or a linked list.

But, in a worst case scenario, all the people are stored in the same bucket, and if the person we are trying to find is last person in this bucket, we need to compare with all the other social security numbers in that bucket before we find the person we are looking for.

In such a worst case scenario the Hash Map has time complexity  $O(n)$ , which is the same time complexity as arrays and linked lists.

To keep Hash Maps fast, it is therefore important to have a hash function that will distribute the entries evenly between the buckets, and to have around as many buckets as Hash Map entries.

Having a lot more buckets than Hash Map entries is a waste of memory, and having a lot less buckets than Hash Map entries is a waste of time.

**Note:** A social security number can be really long, like 11 digits, which means it is possible to store 100 billion people with unique social security numbers. This is a lot more than in any country's population, and even a lot more than there are people on Earth.

buckets can be adjusted to the number of people.

# Hash Map Implementation

Hash Maps in Python are typically done by using Python's own [dictionary data type](#), but to get a better understanding of how Hash Maps work we will not use that here.

To implement a Hash Map in Python we create a class [SimpleHashMap](#).

Inside the [SimpleHashMap](#) class we have a method [\\_\\_init\\_\\_](#) to initialize the Hash Map, a method [hash\\_function](#) for the hash function, and methods for the basic Hash Map operations: [put](#), [get](#), and [remove](#).

We also create a method [print\\_map](#) to better see how the Hash Map looks like.

## Example

```
class SimpleHashMap:
    def __init__(self, size=100):
        self.size = size
        self.buckets = [[] for _ in range(size)] # A list of buckets

    def hash_function(self, key):
        # Sum only the numerical values of the key, ignoring non-numeric
        numeric_sum = sum(int(char) for char in key if char.isdigit())
        return numeric_sum % 10 # Perform modulo 10 on the sum

    def put(self, key, value):
        # Add or update a key-value pair
        index = self.hash_function(key)
        bucket = self.buckets[index]
        for i, (k, v) in enumerate(bucket):
            if k == key:
                bucket[i] = (key, value) # Update existing key
                return
        bucket.append((key, value)) # Add new key-value pair if not found
```

```
26         for k, v in bucket:
27             if k == key:
28                 return v
29         return None # Key not found
30
31     def remove(self, key):
32         # Remove a key-value pair
33         index = self.hash_function(key)
34         bucket = self.buckets[index]
35         for i, (k, v) in enumerate(bucket):
36             if k == key:
37                 del bucket[i] # Remove the key-value pair
38             return
39
40     def print_map(self):
41         # Print all key-value pairs in the hash map
42         print("Hash Map Contents:")
43         for index, bucket in enumerate(self.buckets):
44             print(f"Bucket {index}: {bucket}")
```

Using the `SimpleHashMap` class we can create the same Hash Map as in the top of this page:

## Example

```
print("Updating the name for '123-4570' to 'James'")  
hash_map.put("123-4570", "James")  
  
# Checking if Peter is still there  
print("Name associated with '123-4570':", hash_map.get("123-4570"))
```

[Run Example »](#)[« Previous](#)[Next »](#)

**W3schools Pathfinder**  
Track your progress - it's free! [Sign Up](#) [Log in](#)

ADVERTISEMENT

**Level up your  
skills with our  
courses!**

*Gain skills, and get certified.*

**Start Today!**

# DSA Trees

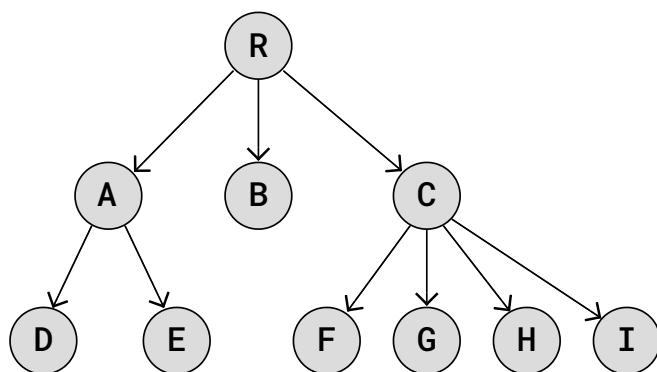
[« Previous](#)[Next »](#)

## Trees

The Tree data structure is similar to [Linked Lists](#) in that each node contains data and can be linked to other nodes.

We have previously covered data structures like Arrays, Linked Lists, Stacks, and Queues. These are all linear structures, which means that each element follows directly after another in a sequence. Trees however, are different. In a Tree, a single element can have multiple 'next' elements, allowing the data structure to branch out in various directions.

The data structure is called a "tree" because it looks like a tree, only upside down, just like in the image below.



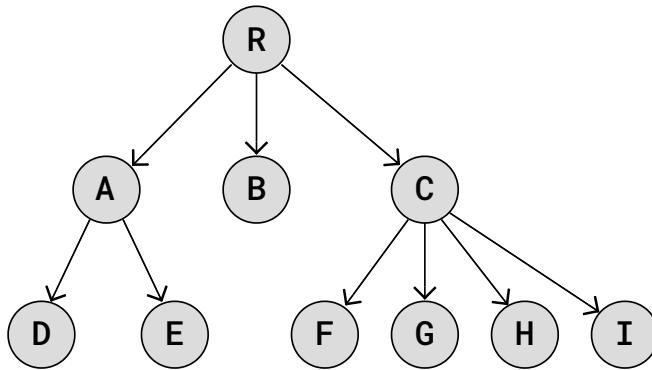
The Tree data structure can be useful in many cases:

- Hierarchical Data: File systems, organizational models, etc.
- Databases: Used for quick data retrieval.
- Routing Tables: Used for routing data in network algorithms.
- Sorting/Search: Used for sorting data and searching for data.

# Tree Terminology and Rules

Learn words used to describe the tree data structure by using the interactive tree visualization below.

- The whole tree
- Root node
- Edges
- Nodes
- Leaf nodes
- Child nodes
- Parent nodes
- Tree height ( $h=2$ )
- Tree size ( $n=10$ )



The first node in a tree is called the **root node**.

A link connecting one node to another is called an **edge**.

A **parent** node has links to its **child** nodes. Another word for a parent node is **internal** node.

A node can have zero, one, or many child nodes.

A node can only have one parent node.

Nodes without links to other child nodes are called **leaves**, or **leaf nodes**.

The **tree height** is the maximum number of edges from the root node to a leaf node. The height of the tree above is 2.

The **height of a node** is the maximum number of edges between the node and a leaf node.

The **tree size** is the number of nodes in the tree.

## Types of Trees

Trees are a fundamental data structure in computer science, used to represent hierarchical relationships. This tutorial covers several key types of trees.

lower value, and the right child node has a higher value.

**AVL Trees:** A type of Binary Search Tree that self-balances so that for every node, the difference in height between the left and right subtrees is at most one. This balance is maintained through rotations when nodes are inserted or deleted.

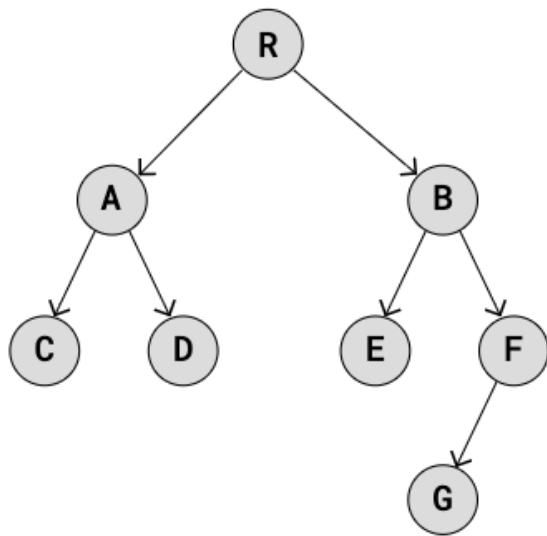
Each of these data structures are described in detail on the next pages, including animations and how to implement them.

## DSA Exercises

### Test Yourself With Exercises

#### Exercise:

In a Tree data structure, like the one below:



What are nodes C, D, E, and G called?

Nodes C, D, E, and G

# DSA Binary Trees

[« Previous](#)[Next »](#)

## Binary Trees

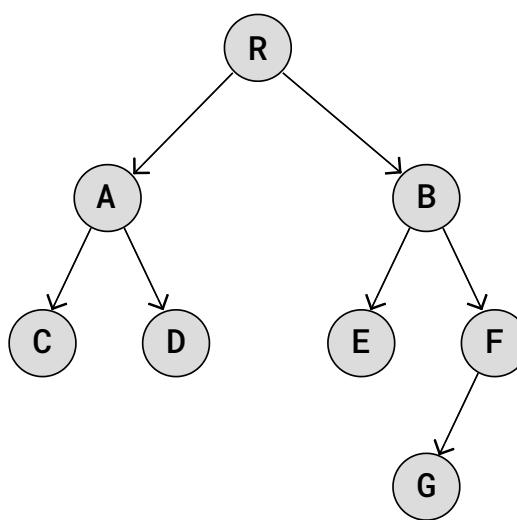
A Binary Tree is a type of tree data structure where each node can have a maximum of two child nodes, a left child node and a right child node.

This restriction, that a node can have a maximum of two child nodes, gives us many benefits:

- Algorithms like traversing, searching, insertion and deletion become easier to understand, to implement, and run faster.
- Keeping data sorted in a Binary Search Tree (BST) makes searching very efficient.
- Balancing trees is easier to do with a limited number of child nodes, using an AVL Binary Tree for example.
- Binary Trees can be represented as arrays, making the tree more memory efficient.

Use the animation below to see how a Binary Tree looks, and what words we use to describe it.

- The Binary Tree
- Root node
- A's left child
- A's right child
- B's subtree
- Tree size (n=8)
- Tree height (h=3)
- Child nodes
- Parent/internal nodes



The right child node is the child node to the right.

The **tree height** is the maximum number of edges from the root node to a leaf node.

## Binary Trees vs Arrays and Linked Lists

Benefits of Binary Trees over Arrays and Linked Lists:

- **Arrays** are fast when you want to access an element directly, like element number 700 in an array of 1000 elements for example. But inserting and deleting elements require other elements to shift in memory to make place for the new element, or to take the deleted elements place, and that is time consuming.
- **Linked Lists** are fast when inserting or deleting nodes, no memory shifting needed, but to access an element inside the list, the list must be traversed, and that takes time.
- **Binary Trees**, such as Binary Search Trees and AVL Trees, are great compared to Arrays and Linked Lists because they are BOTH fast at accessing a node, AND fast when it comes to deleting or inserting a node, with no shifts in memory needed.

We will take a closer look at how Binary Search Trees (BSTs) and AVL Trees work on the next two pages, but first let's look at how a Binary Tree can be implemented, and how it can be traversed.

## Types of Binary Trees

There are different variants, or types, of Binary Trees worth discussing to get a better understanding of how Binary Trees can be structured.

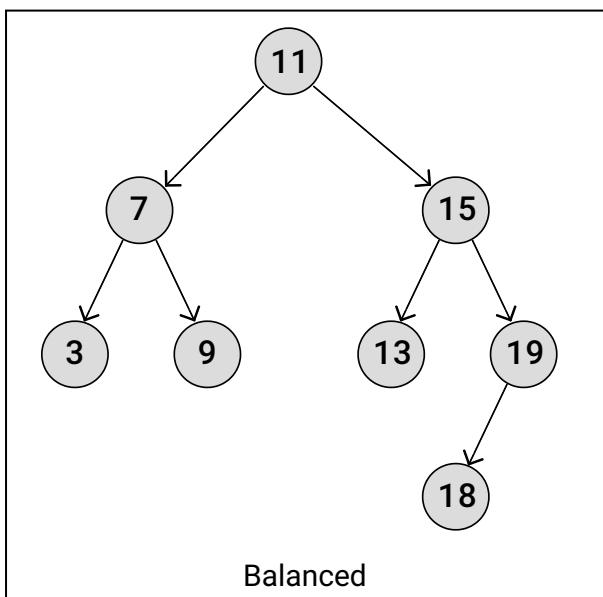
The different kinds of Binary Trees are also worth mentioning now as these words and concepts will be used later in the tutorial.

Below are short explanations of different types of Binary Tree structures, and below the explanations are drawings of these kinds of structures to make it as easy to understand as possible.

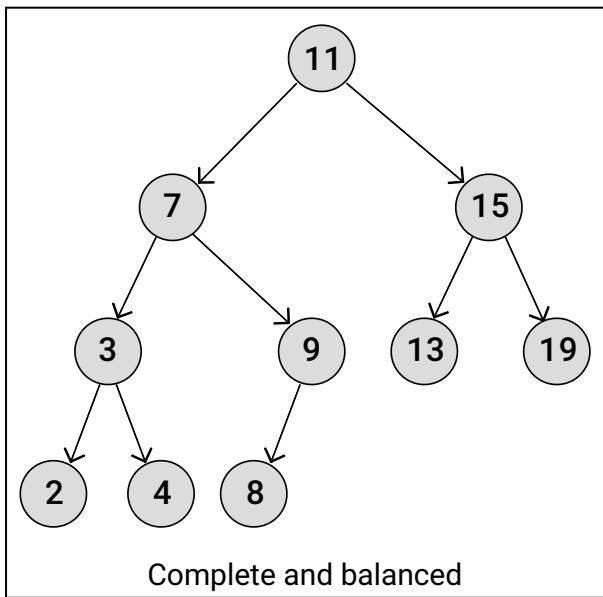
A **balanced** Binary Tree has at most 1 in difference between its left and right subtree heights, for each node in the tree.

A **complete** Binary Tree has all levels full of nodes, except the last level, which is can also be full, or filled from left to right. The properties of a complete Binary Tree means it is also balanced.

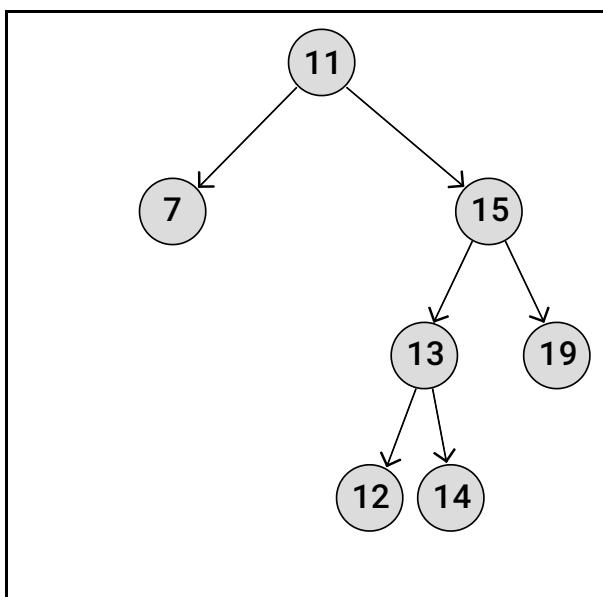
A **full** Binary Tree is a kind of tree where each node has either 0 or 2 child nodes.

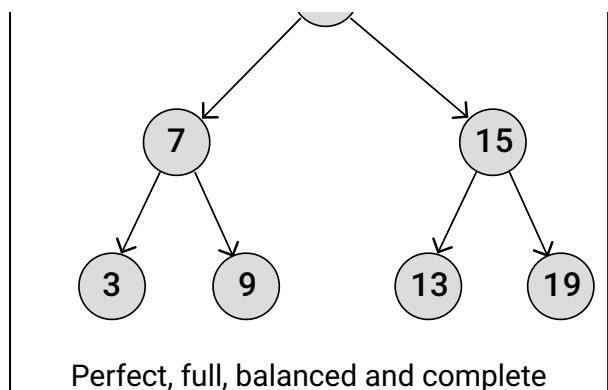


Balanced



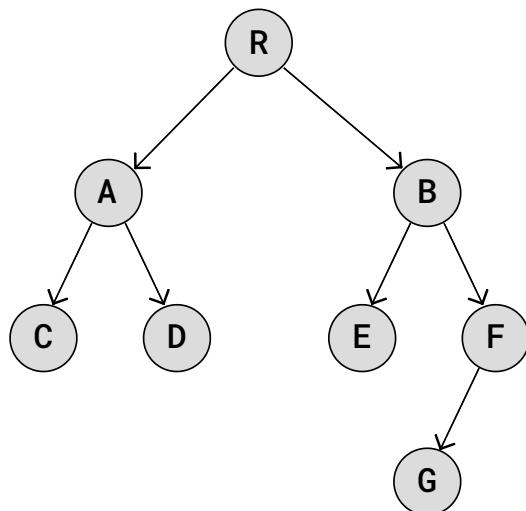
Complete and balanced





## Binary Tree Implementation

Let's implement this Binary Tree:



The Binary Tree above can be implemented much like we implemented a [Singly Linked List](#), except that instead of linking each node to one next node, we create a structure where each node can be linked to both its left and right child nodes.

This is how a Binary Tree can be implemented:

### Example

Python:

```
class TreeNode:  
    def __init__(self, data):
```

```
root = TreeNode('R')
nodeA = TreeNode('A')
nodeB = TreeNode('B')
nodeC = TreeNode('C')
nodeD = TreeNode('D')
nodeE = TreeNode('E')
nodeF = TreeNode('F')
nodeG = TreeNode('G')

root.left = nodeA
root.right = nodeB

nodeA.left = nodeC
nodeA.right = nodeD

nodeB.left = nodeE
nodeB.right = nodeF

nodeF.left = nodeG

# Test
print("root.right.left.data:", root.right.left.data)
```

[Run Example »](#)

## Binary Tree Traversal

Going through a Tree by visiting every node, one node at a time, is called traversal.

Since Arrays and Linked Lists are linear data structures, there is only one obvious way to traverse these: start at the first element, or node, and continue to visit the next until you have visited them all.

But since a Tree can branch out in different directions (non-linear), there are different ways of traversing Trees.

There are two main categories of Tree traversal methods:

exploring the tree branch by branch in a downwards direction.

There are three different types of DFS traversals:

- pre-order
- in-order
- post-order

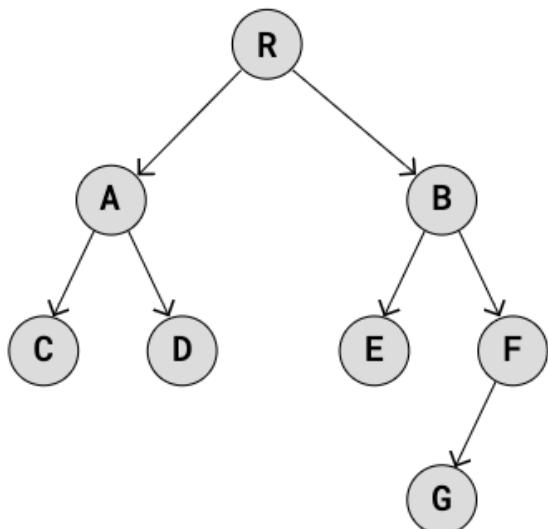
These three Depth First Search traversals are described in detail on the next pages.

## DSA Exercises

## Test Yourself With Exercises

### Exercise:

In a Binary Tree data structure, like the one below:



What is the relationship between node B and nodes E and F?

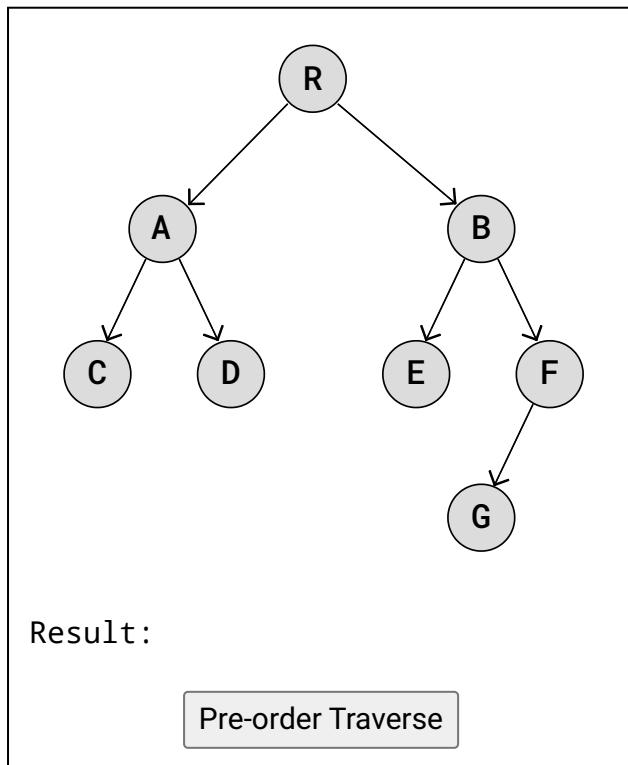
# DSA Pre-order Traversal

[« Previous](#)[Next »](#)

## Pre-order Traversal of Binary Trees

Pre-order Traversal is a type of Depth First Search, where each node is visited in a certain order. Read more about Binary Tree traversals in general [here](#).

Pre-order traversal of a Binary Tree looks like this:



Pre-order Traversal is done by visiting the root node first, then recursively do a pre-order traversal of the left subtree, followed by a recursive pre-order traversal of the right subtree. It's used for creating a copy of the tree, prefix notation of an expression tree, etc.

## Example

Python:

```
1 | def preOrderTraversal(node):
2 |     if node is None:
3 |         return
4 |     print(node.data, end=", ")
5 |     preOrderTraversal(node.left)
6 |     preOrderTraversal(node.right)
```

[Run Example »](#)

The first node to be printed is node R, as the Pre-order Traversal works by first visiting, or printing, the current node (line 4), before calling the left and right child nodes recursively (line 5 and 6).

The `preOrderTraversal()` function keeps traversing the left subtree recursively (line 5), before going on to traversing the right subtree (line 6). So the next nodes that are printed are 'A' and then 'C'.

The first time the argument `node is None` is when the left child of node C is given as an argument (C has no left child).

After `None` is returned the first time when calling C's left child, C's right child also returns `None`, and then the recursive calls continue to propagate back so that A's right child D is the next to be printed.

The code continues to propagate back so that the rest of the nodes in R's right subtree gets printed.

[« Previous](#)

[Next »](#)

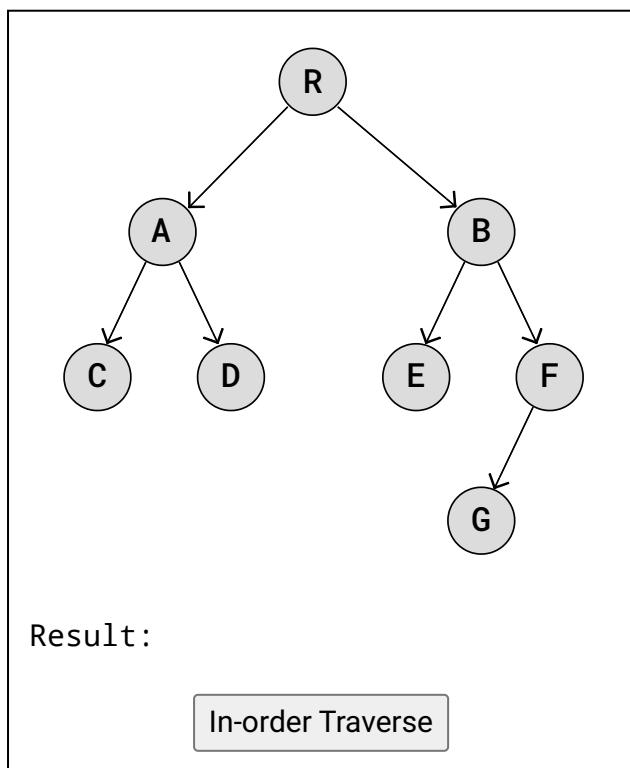
# DSA In-order Traversal

[« Previous](#)[Next »](#)

## In-order Traversal of Binary Trees

In-order Traversal is a type of Depth First Search, where each node is visited in a certain order. Read more about Binary Tree traversals in general [here](#).

Run the animation below to see how an In-order Traversal of a Binary Tree is done.



In-order Traversal does a recursive In-order Traversal of the left subtree, visits the root node, and finally, does a recursive In-order Traversal of the right subtree. This traversal is mainly used for Binary Search Trees where it returns values in ascending order.

This is how the code for In-order Traversal looks like:

## Example

Python:

```
1 | def inOrderTraversal(node):
2 |     if node is None:
3 |         return
4 |     inOrderTraversal(node.left)
5 |     print(node.data, end=", ")
6 |     inOrderTraversal(node.right)
```

[Run Example »](#)

The `inOrderTraversal()` function keeps calling itself with the current left child node as an argument (line 4) until that argument is `None` and the function returns (line 2-3).

The first time the argument `node` is `None` is when the left child of node C is given as an argument (C has no left child).

After that, the `data` part of node C is printed (line 5), which means that 'C' is the first thing that gets printed.

Then, node C's right child is given as an argument (line 6), which is `None`, so the function call returns without doing anything else.

After 'C' is printed, the previous `inOrderTraversal()` function calls continue to run, so that 'A' gets printed, then 'D', then 'R', and so on.

[« Previous](#)

[Next »](#)

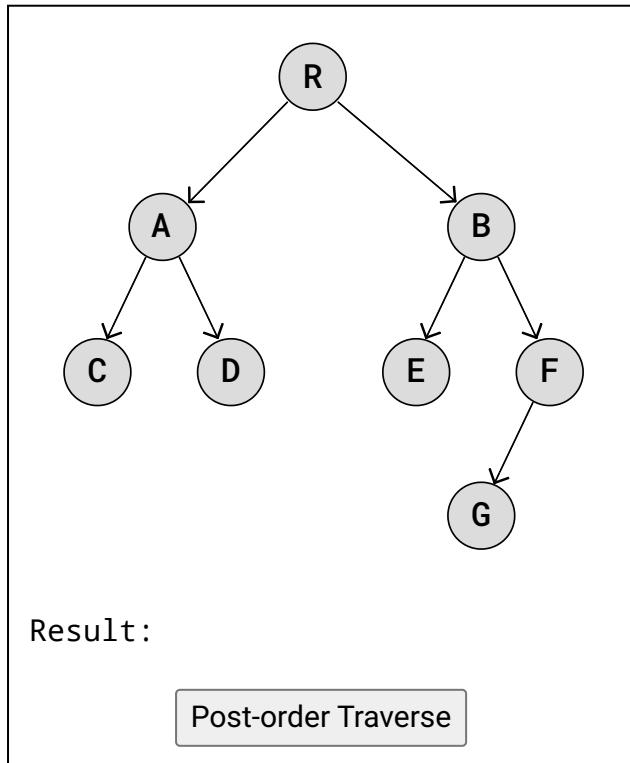
# DSA Post-order Traversal

[« Previous](#)[Next »](#)

## Post-order Traversal of Binary Trees

Post-order Traversal is a type of Depth First Search, where each node is visited in a certain order. Read more about Binary Tree traversals in general [here](#).

Doing a Post-order Traversal on a Binary Tree can be visualized like this:



Post-order Traversal works by recursively doing a Post-order Traversal of the left subtree and the right subtree, followed by a visit to the root node. It is used for deleting a tree, post-fix notation of an expression tree, etc.

## Example

Python:

```
1 | def postOrderTraversal(node):
2 |     if node is None:
3 |         return
4 |     postOrderTraversal(node.left)
5 |     postOrderTraversal(node.right)
6 |     print(node.data, end=", ")
```

[Run Example »](#)

The `postOrderTraversal()` function keeps traversing the left subtree recursively (line 4), until `None` is returned when C's left child node is called as the `node` argument.

After C's left child node returns `None`, line 5 runs and C's right child node returns `None`, and then the letter 'C' is printed (line 6).

This means that C is visited, or printed, "after" its left and right child nodes are traversed, that is why it is called "post" order traversal.

The `postOrderTraversal()` function continues to propagate back to previous recursive function calls, so the next node to be printed is 'D', then 'A'.

The function continues to propagate back and printing nodes until all nodes are printed, or visited.

[« Previous](#)

[Next »](#)

# DSA Array Implementation

[« Previous](#)[Next »](#)

## Array Implementation of Binary Trees

To avoid the cost of all the shifts in memory that we get from using Arrays, it is useful to implement Binary Trees with pointers from one element to the next, just like Binary Trees are implemented before this point, especially when the Binary Tree is modified often.

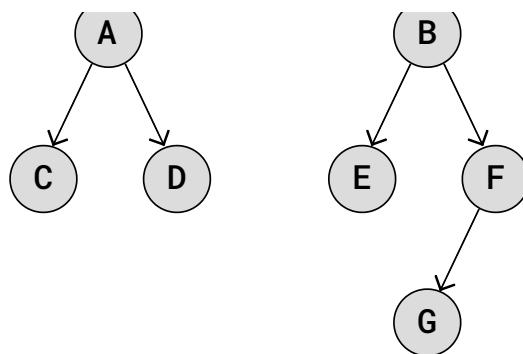
But in case we read from the Binary Tree a lot more than we modify it, an Array implementation of a Binary Tree can make sense as it needs less memory, it can be easier to implement, and it can be faster for certain operations due to cache locality.

**Cache Locality** is when the fast cache memory in the computer stores parts of memory that was recently accessed, or when the cache stores parts of memory that is close to the address that is currently accessed. This happens because it is likely that the CPU needs something in the next cycle that is close to what it used in the previous cycle, either close in time or close in space.

Since Array elements are stored contiguously in memory, one element right after the other, computers are sometimes faster when reading from Arrays because the next element is already cached, available for fast access in case the CPU needs it in the next cycle.

How arrays are stored in memory is explained more in detail [here](#).

Consider this Binary Tree:



This Binary Tree can be stored in an Array starting with the root node R on index 0. The rest of the tree can be built by taking a node stored on index  $i$ , and storing its left child node on index  $2 \cdot i + 1$ , and its right child node on index  $2 \cdot i + 2$ .

Below is an Array implementation of the Binary Tree.

## Example

Python:

```
1  binary_tree_array = ['R', 'A', 'B', 'C', 'D', 'E', 'F', None, None
2
3  def left_child_index(index):
4      return 2 * index + 1
5
6  def right_child_index(index):
7      return 2 * index + 2
8
9  def get_data(index):
10     if 0 <= index < len(binary_tree_array):
11         return binary_tree_array[index]
12     return None
13
14 right_child = right_child_index(0)
15 left_child_of_right_child = left_child_index(right_child)
16 data = get_data(left_child_of_right_child)
17
18 print("root.right.left.data:", data)
```

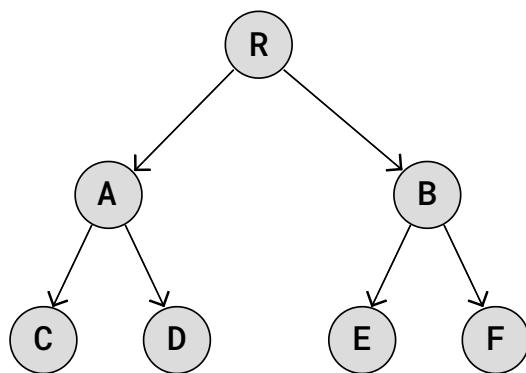
[Run Example »](#)

child is on index  $2 \cdot 2 + 1 = 5$ , which is node E, right? And B's right child is on index  $2 \cdot 2 + 2 = 6$ , which is node F, and that also fits with the drawing above, right?

As you can see on line 1, this implementation requires empty array elements where nodes have no child nodes. So to avoid wasting space on empty Array elements, Binary Trees stored using Array implementation should be a "perfect" Binary Tree, or a nearly perfect one.

A perfect Binary Tree is when every internal node have exactly two child nodes, and all leaf nodes are on the same level.

If we remove the G node in the Binary Tree above, it looks like this:



And the first line in the code above can be written without wasting space on empty Array elements:

```
1 | binary_tree_array = ['R', 'A', 'B', 'C', 'D', 'E', 'F']
```

This is how the three different DFS traversals can be done on an Array implementation of a Binary Tree.

## Example

Python:

```
binary_tree_array = ['R', 'A', 'B', 'C', 'D', 'E', 'F', None, None

def left_child_index(index):
    return 2 * index + 1
```

```
1 def pre_order(index):
2     if index >= len(binary_tree_array) or binary_tree_array[index] == None:
3         return []
4     return [binary_tree_array[index]] + pre_order(left_child_index(index))
5
6 def in_order(index):
7     if index >= len(binary_tree_array) or binary_tree_array[index] == None:
8         return []
9     return in_order(left_child_index(index)) + [binary_tree_array[index]] + in_order(right_child_index(index))
10
11 def post_order(index):
12     if index >= len(binary_tree_array) or binary_tree_array[index] == None:
13         return []
14     return post_order(left_child_index(index)) + post_order(right_child_index(index)) + [binary_tree_array[index]]
15
16 print("Pre-order Traversal:", pre_order(0))
17 print("In-order Traversal:", in_order(0))
18 print("Post-order Traversal:", post_order(0))
```

[Run Example »](#)

By comparing how these traversals are done in an array implementation to how the pointer implementation was traversed, you can see that the pre-order, in-order, and post-order traversals works in the same recursive way.

[« Previous](#)[Next »](#)[W3schools Pathfinder](#)[Track your progress - it's free!](#)[Sign Up](#)[Log in](#)

# DSA Binary Search Trees

[« Previous](#)[Next »](#)

A **Binary Search Tree** is a Binary Tree where every node's left child has a lower value, and every node's right child has a higher value.

A clear advantage with Binary Search Trees is that operations like search, delete, and insert are fast and done without having to shift values in memory.

## Binary Search Trees

A Binary Search Tree (BST) is a type of [Binary Tree data structure](#), where the following properties must be true for any node "X" in the tree:

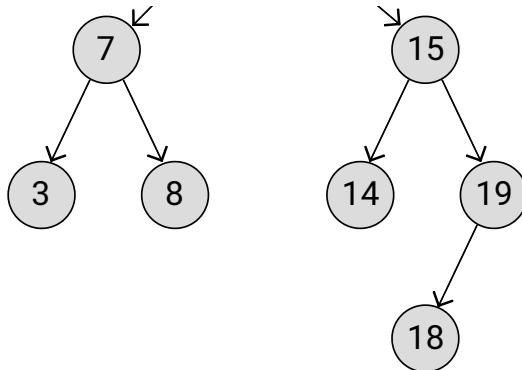
- The X node's left child and all of its descendants (children, children's children, and so on) have lower values than X's value.
- The right child, and all its descendants have higher values than X's value.
- Left and right subtrees must also be Binary Search Trees.

These properties makes it faster to search, add and delete values than a regular binary tree.

To make this as easy to understand and implement as possible, let's also assume that all values in a Binary Search Tree are unique.

Use the Binary Search Tree below to better understand these concepts and relevant terminology.

- 7's left child
- 7's right child
- Tree height ( $h=3$ )
- 15's height ( $h=2$ )
- 13's right subtree
- 13's in-order successor
- Child nodes
- Parent/Internal nodes
- Leaf nodes



The **size** of a tree is the number of nodes in it ( $n$ ).

A **subtree** starts with one of the nodes in the tree as a local root, and consists of that node and all its descendants.

The **descendants** of a node are all the child nodes of that node, and all their child nodes, and so on. Just start with a node, and the descendants will be all nodes that are connected below that node.

The **node's height** is the maximum number of edges between that node and a leaf node.

A **node's in-order successor** is the node that comes after it if we were to do in-order traversal. In-order traversal of the BST above would result in node 13 coming before node 14, and so the successor of node 13 is node 14.

## Traversal of a Binary Search Tree

Just to confirm that we actually have a Binary Search Tree data structure in front of us, we can check if the properties at the top of this page are true. So for every node in the figure above, check if all the values to the left of the node are lower, and that all values to the right are higher.

Another way to check if a Binary Tree is BST, is to do an in-order traversal (like we did on the previous page) and check if the resulting list of values are in an increasing order.

The code below is an implementation of the Binary Search Tree in the figure above, with traversal.

## Example

☰ TLIN SASS VUE DSA GEN AI SCIPY AWS CYBERSECURITY DATA SC

```
self.data = data
self.left = None
self.right = None

def inOrderTraversal(node):
    if node is None:
        return
    inOrderTraversal(node.left)
    print(node.data, end=", ")
    inOrderTraversal(node.right)

root = TreeNode(13)
node7 = TreeNode(7)
node15 = TreeNode(15)
node3 = TreeNode(3)
node8 = TreeNode(8)
node14 = TreeNode(14)
node19 = TreeNode(19)
node18 = TreeNode(18)

root.left = node7
root.right = node15

node7.left = node3
node7.right = node8

node15.left = node14
node15.right = node19

node19.left = node18

# Traverse
inOrderTraversal(root)
```

[Run Example »](#)

As we can see by running the code example above, the in-order traversal produces a list of numbers in an increasing (ascending) order, which means that this Binary Tree is a Binary Search Tree.

array.

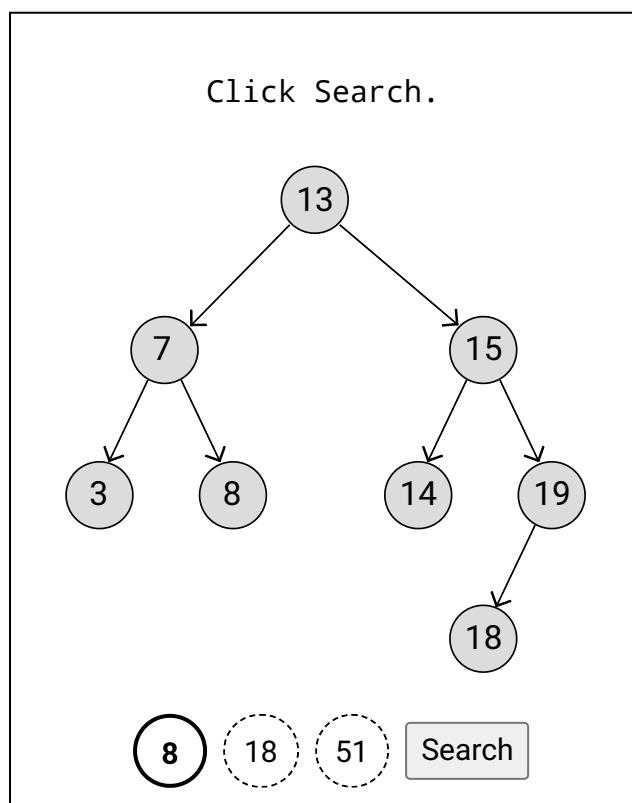
For Binary Search to work, the array must be sorted already, and searching for a value in an array can then be done really fast.

Similarly, searching for a value in a BST can also be done really fast because of how the nodes are placed.

### How it works:

1. Start at the root node.
2. If this is the value we are looking for, return.
3. If the value we are looking for is higher, continue searching in the right subtree.
4. If the value we are looking for is lower, continue searching in the left subtree.
5. If the subtree we want to search does not exist, depending on the programming language, return `None`, or `NULL`, or something similar, to indicate that the value is not inside the BST.

Use the animation below to see how we search for a value in a Binary Search Tree.



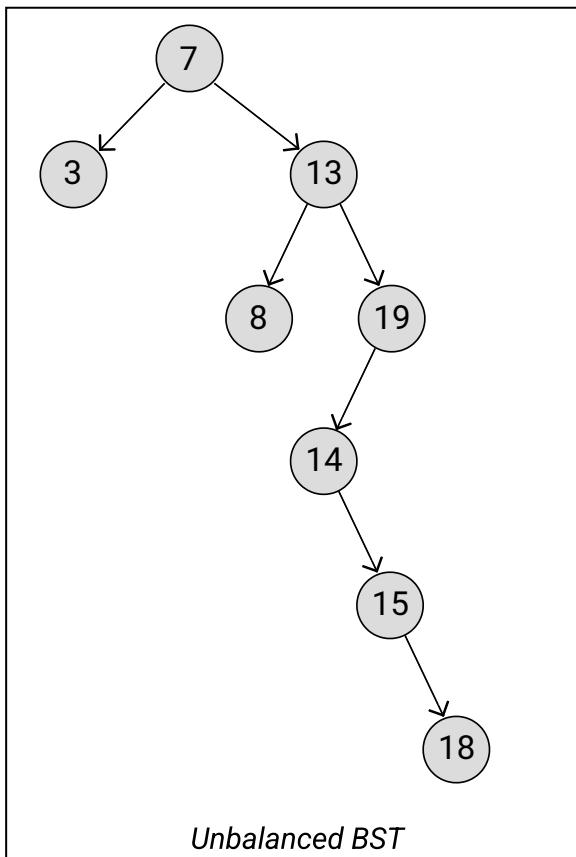
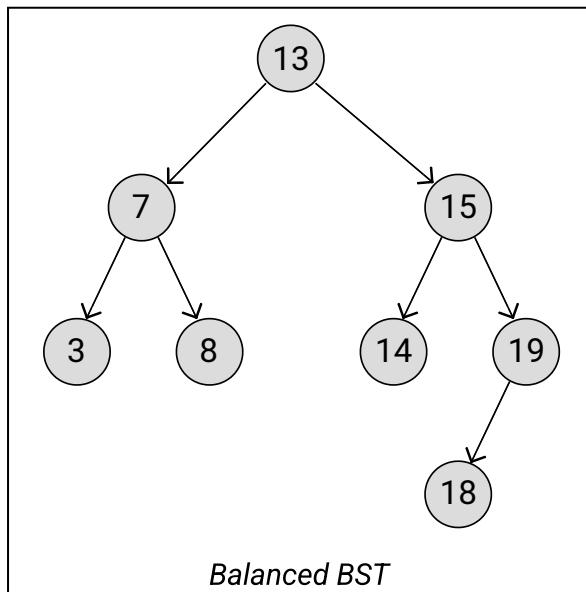
Python:

```
def search(node, target):
    if node is None:
        return None
    elif node.data == target:
        return node
    elif target < node.data:
        return search(node.left, target)
    else:
        return search(node.right, target)
```

[Run Example »](#)

The time complexity for searching a BST for a value is  $O(h)$ , where  $h$  is the height of the tree.

For a BST with most nodes on the right side for example, the height of the tree becomes larger than it needs to be, and the worst case search will take longer. Such trees are called unbalanced.



We will use the next page to describe a type of Binary Tree called AVL Trees. AVL trees are self-balancing, which means that the height of the tree is kept to a minimum so that operations like search, insertion and deletion take less time.

# Insert a Node in a BST

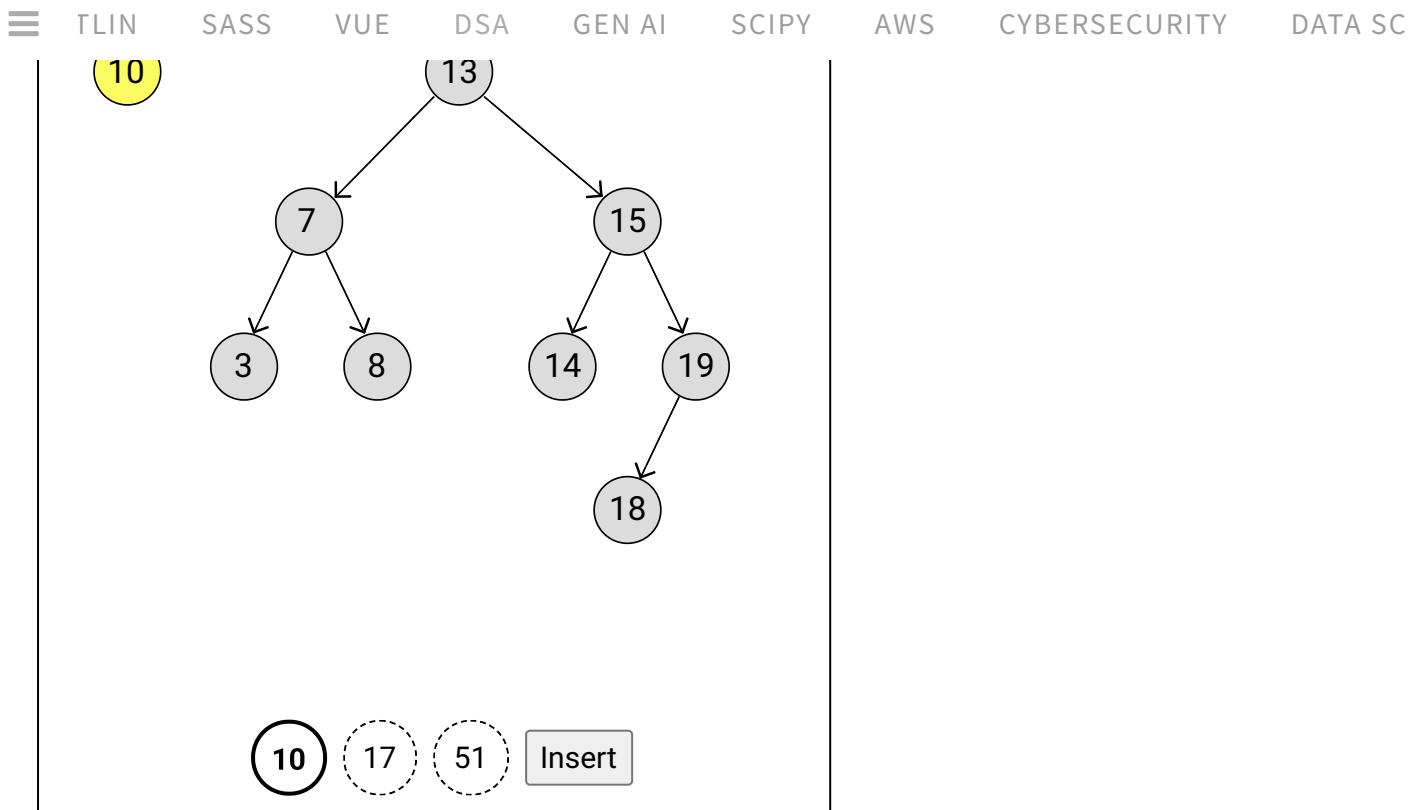
Inserting a node in a BST is similar to searching for a value.

## How it works:

1. Start at the root node.
2. Compare each node:
  - Is the value lower? Go left.
  - Is the value higher? Go right.
3. Continue to compare nodes with the new value until there is no right or left to compare with.  
That is where the new node is inserted.

Inserting nodes as described above means that an inserted node will always become a new leaf node.

Use the simulation below to see how new nodes are inserted.



All nodes in the BST are unique, so in case we find the same value as the one we want to insert, we do nothing.

This is how node insertion in BST can be implemented:

## Example

Python:

```
def insert(node, data):
    if node is None:
        return TreeNode(data)
    else:
        if data < node.data:
            node.left = insert(node.left, data)
        elif data > node.data:
            node.right = insert(node.right, data)
    return node
```

[Run Example »](#)

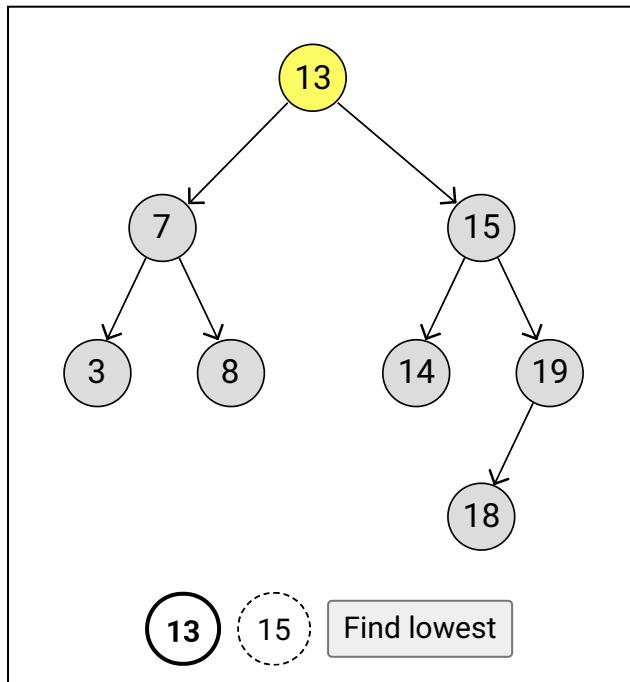
that finds the lowest value in a node's subtree.

### How it works:

1. Start at the root node of the subtree.
2. Go left as far as possible.
3. The node you end up in is the node with the lowest value in that BST subtree.

In the figure below, if we start at node 13 and keep going left, we end up in node 3, which is the lowest value, right?

And if we start at node 15 and keep going left, we end up in node 14, which is the lowest value in node 15's subtree.



This is how the function for finding the lowest value in the subtree of a BST node looks like:

## Example

Python:

```
current = current.left  
return current
```

[Run Example »](#)

We will use this `minValueNode()` function in the section below, to find a node's in-order successor, and use that to delete a node.

## Delete a Node in a BST

To delete a node, our function must first search the BST to find it.

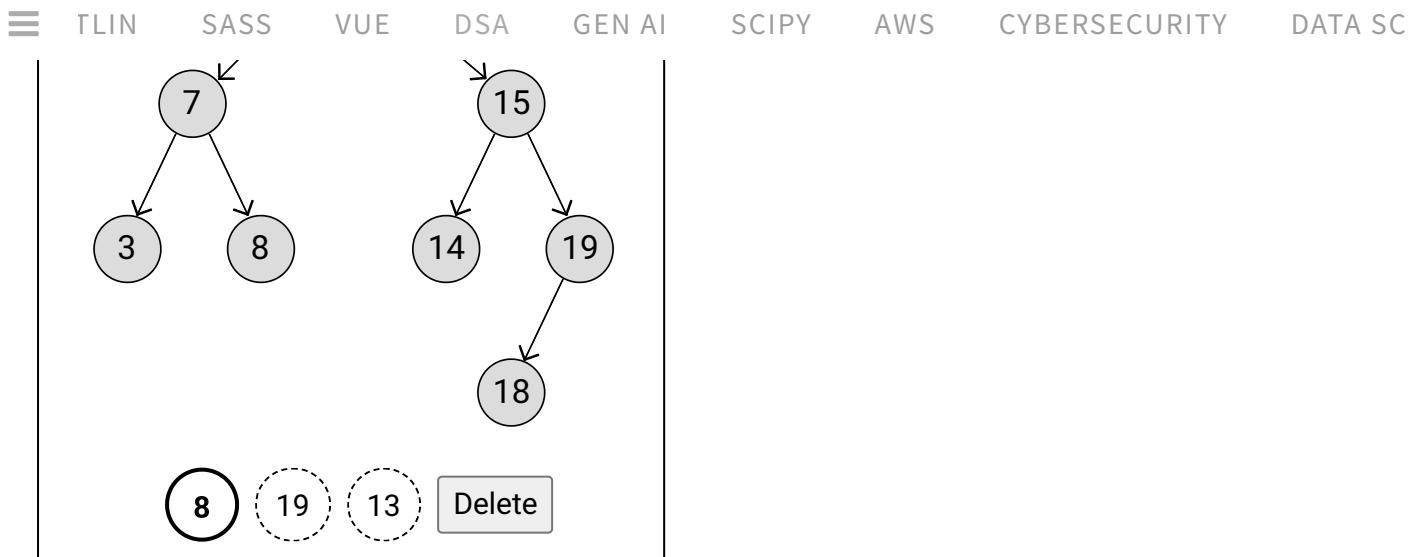
After the node is found there are three different cases where deleting a node must be done differently.

### How it works:

1. If the node is a leaf node, remove it by removing the link to it.
2. If the node only has one child node, connect the parent node of the node you want to remove to that child node.
3. If the node has both right and left child nodes: Find the node's in-order successor, change values with that node, then delete it.

In step 3 above, the successor we find will always be a leaf node, and because it is the node that comes right after the node we want to delete, we can swap values with it and delete it.

Use the animation below to see how different nodes are deleted.



**Node 8** is a leaf node (case 1), so after we find it, we can just delete it.

**Node 19** has only one child node (case 2). To delete node 19, the parent node 15 is connected directly to node 18, and then node 19 can be removed.

**Node 13** has two child nodes (case 3). We find the successor, the node that comes right after during in-order traversal, by finding the lowest node in node 13's right subtree, which is node 14. Value 14 is put into node 13, and then we can delete node 14.

This is how a BST can be implemented with functionality for deleting a node:

## Example

Python:

```
1 def delete(node, data):
2     if not node:
3         return None
4
5     if data < node.data:
6         node.left = delete(node.left, data)
7     elif data > node.data:
8         node.right = delete(node.right, data)
9     else:
10        # Node with only one child or no child
11        if not node.left:
12            temp = node.right
13            node = None
14            return temp
15
```

```
--  
20      # Node with two children, get the in-order successor  
21      node.data = minValueNode(node.right).data  
22      node.right = delete(node.right, node.data)  
23  
24  return node
```

[Run Example »](#)

**Line 1:** The `node` argument here makes it possible for the function to call itself recursively on smaller and smaller subtrees in the search for the node with the `data` we want to delete.

**Line 2-8:** This is searching for the node with correct `data` that we want to delete.

**Line 9-22:** The node we want to delete has been found. There are three such cases:

**Case 1:** Node with no child nodes (leaf node). `None` is returned, and that becomes the parent node's new left or right value by recursion (line 6 or 8).

**Case 2:** Node with either left or right child node. That left or right child node becomes the parent's new left or right child through recursion (line 7 or 9).

**Case 3:** Node has both left and right child nodes. The in-order successor is found using the `minValueNode()` function. We keep the successor's value by setting it as the value of the node we want to delete, and then we can delete the successor node.

**Line 24:** `node` is returned to maintain the recursive functionality.

## BST Compared to Other Data Structures

Binary Search Trees take the best from two other data structures: Arrays and Linked Lists.

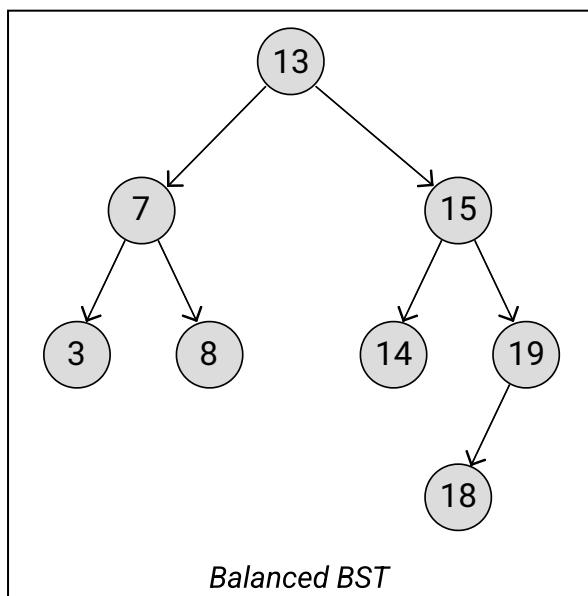
Data Structure	Searching for a value	Delete / Insert leads to shifting in memory
Sorted Array	$O(\log n)$	Yes
Linked List	$O(n)$	No
Binary Search Tree	$O(\log n)$	No

with Linked Lists.

# BST Balance and Time Complexity

On a Binary Search Tree, operations like inserting a new node, deleting a node, or searching for a node are actually  $O(h)$ . That means that the higher the tree is ( $h$ ), the longer the operation will take.

The reason why we wrote that searching for a value is  $O(\log n)$  in the table above is because that is true if the tree is "balanced", like in the image below.

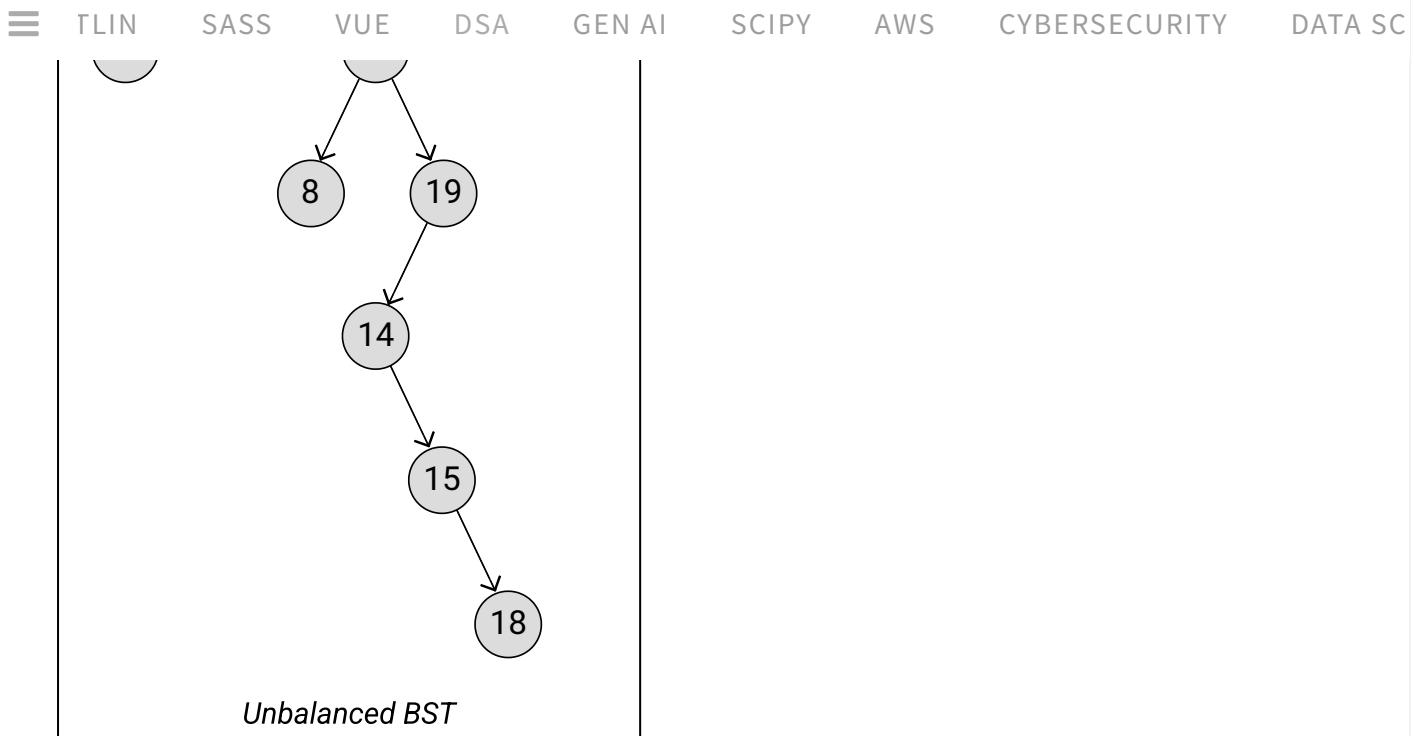


We call this tree balanced because there are approximately the same number of nodes on the left and right side of the tree.

The exact way to tell that a Binary Tree is balanced is that the height of the left and right subtrees of any node only differs by one. In the image above, the left subtree of the root node has height  $h = 2$ , and the right subtree has height  $h = 3$ .

For a balanced BST, with a large number of nodes (big  $n$ ), we get height  $h \approx \log_2 n$ , and therefore the time complexity for searching, deleting, or inserting a node can be written as  $O(h) = O(\log n)$ .

But, in case the BST is completely unbalanced, like in the image below, the height of the tree is approximately the same as the number of nodes,  $h \approx n$ , and we get time complexity  $O(h) = O(n)$  for searching, deleting, or inserting a node.



So, to optimize operations on a BST, the height must be minimized, and to do that the tree must be balanced.

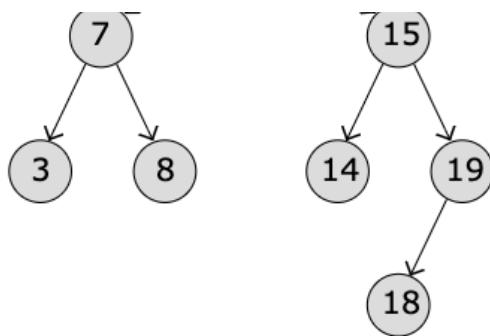
And keeping a Binary Search Tree balanced is exactly what AVL Trees do, which is the data structure explained on the next page.

## DSA Exercises

Test Yourself With Exercises

### Exercise:

Inserting a node with value 6 in this Binary Search Tree:



Where is the new node inserted?

The node with value 6 becomes the right child node of the node with value .

[Submit Answer »](#)

[Start the Exercise](#)

[« Previous](#)

[Next »](#)

**W3schools Pathfinder**  
Track your progress - it's free!

[Sign Up](#) [Log in](#)

# DSA AVL Trees

[« Previous](#)[Next »](#)

The **AVL Tree** is a type of Binary Search Tree named after two Soviet inventors Georgy Adelson-Velsky and Evgenii Landis who invented the AVL Tree in 1962.

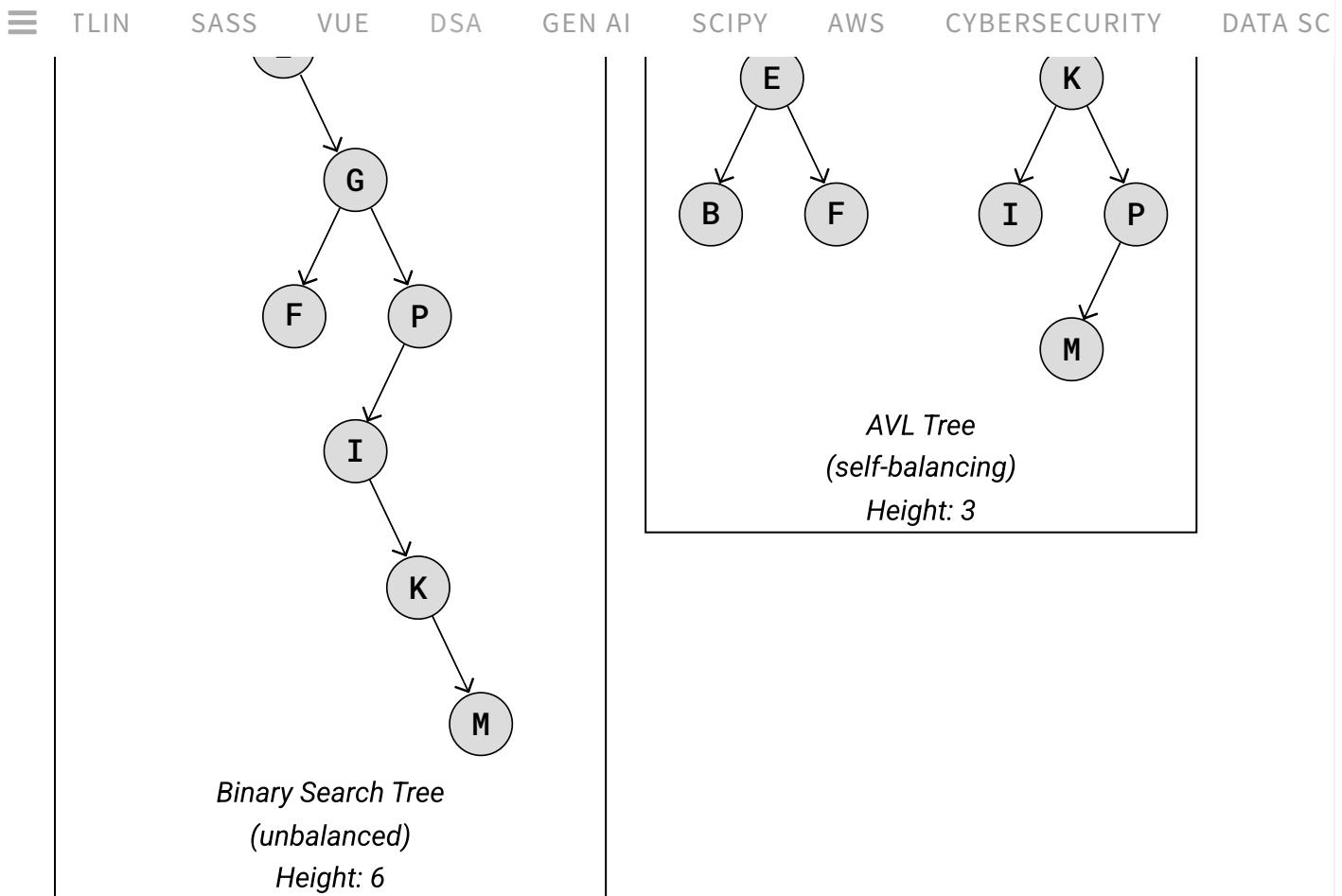
AVL trees are self-balancing, which means that the tree height is kept to a minimum so that a very fast runtime is guaranteed for searching, inserting and deleting nodes, with time complexity  $O(\log n)$ .

## AVL Trees

The only difference between a regular [Binary Search Tree](#) and an AVL Tree is that AVL Trees do rotation operations in addition, to keep the tree balance.

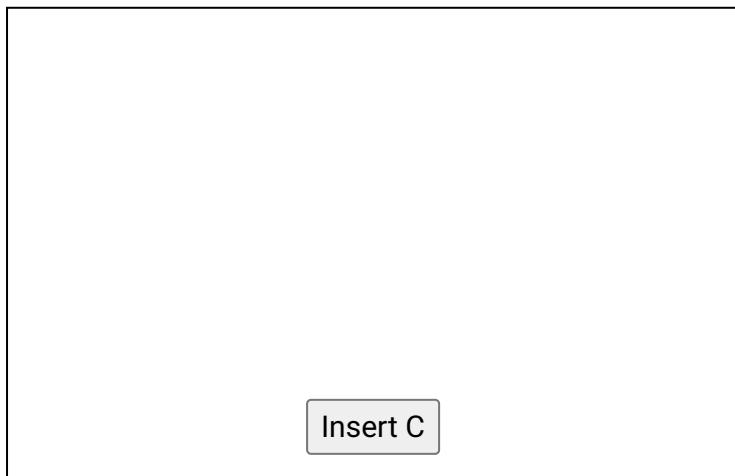
A Binary Search Tree is in balance when the difference in height between left and right subtrees is less than 2.

By keeping balance, the AVL Tree ensures a minimum tree height, which means that search, insert, and delete operations can be done really fast.



The two trees above are both Binary Search Trees, they have the same nodes, and the same in-order traversal (alphabetical), but the height is very different because the AVL Tree has balanced itself.

Step through the building of an AVL Tree in the animation below to see how the balance factors are updated, and how rotation operations are done when required to restore the balance.

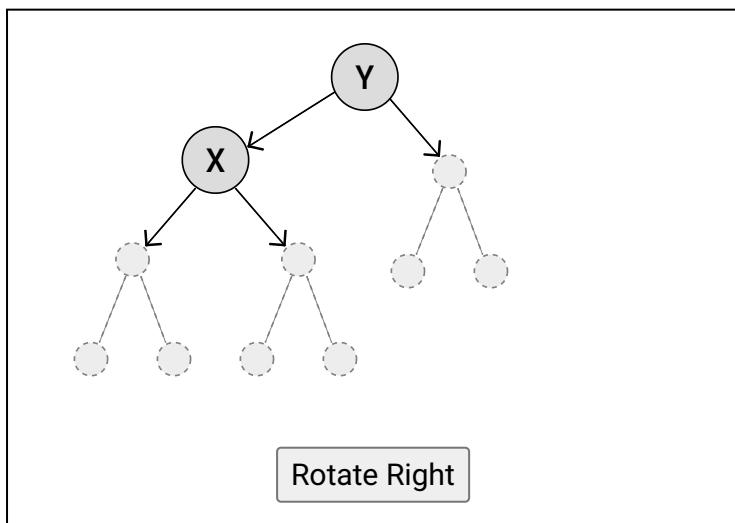


# Left and Right Rotations

To restore balance in an AVL Tree, left or right rotations are done, or a combination of left and right rotations.

The previous animation shows one specific left rotation, and one specific right rotation.

But in general, left and right rotations are done like in the animation below.



Notice how the subtree changes its parent. Subtrees change parent in this way during rotation to maintain the correct in-order traversal, and to maintain the BST property that the left child is less than the right child, for all nodes in the tree.

Also keep in mind that it is not always the root node that become unbalanced and need rotation.

## The Balance Factor

A node's balance factor is the difference in subtree heights.

The subtree heights are stored at each node for all nodes in an AVL Tree, and the balance factor is calculated based on its subtree heights to check if the tree has become out of balance.

The height of a subtree is the number of edges between the root node of the subtree and the leaf node farthest down in that subtree.

$$BF(X) = \text{height}(\text{rightSubtree}(X)) - \text{height}(\text{leftSubtree}(X))$$

Balance factor values

- 0: The node is in balance.
- more than 0: The node is "right heavy".
- less than 0: The node is "left heavy".

If the balance factor is less than -1, or more than 1, for one or more nodes in the tree, the tree is considered not in balance, and a rotation operation is needed to restore balance.

Let's take a closer look at the different rotation operations that an AVL Tree can do to regain balance.

## The Four "out-of-balance" Cases

When the balance factor of just one node is less than -1, or more than 1, the tree is regarded as out of balance, and a rotation is needed to restore balance.

There are four different ways an AVL Tree can be out of balance, and each of these cases require a different rotation operation.

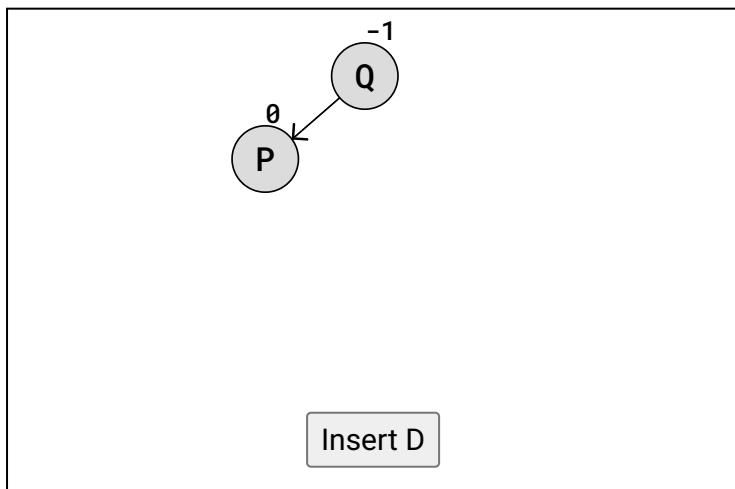
Case	Description	Rotation to Restore Balance
Left-Left (LL)	The unbalanced node and its left child node are both left-heavy.	A single right rotation.
Right-Right (RR)	The unbalanced node and its right child node are both right-heavy.	A single left rotation.
Left-Right (LR)	The unbalanced node is left heavy, and its left child node is right heavy.	First do a left rotation on the left child node, then do a right rotation on the unbalanced node.
Right-Left (RL)	The unbalanced node is right heavy, and its right child node is left heavy.	First do a right rotation on the right child node, then do a left rotation on the unbalanced node.

## THE LEFT-LEFT (LL) CASE

The node where the unbalance is discovered is left heavy, and the node's left child node is also left heavy.

When this LL case happens, a single right rotation on the unbalanced node is enough to restore balance.

Step through the animation below to see the LL case, and how the balance is restored by a single right rotation.



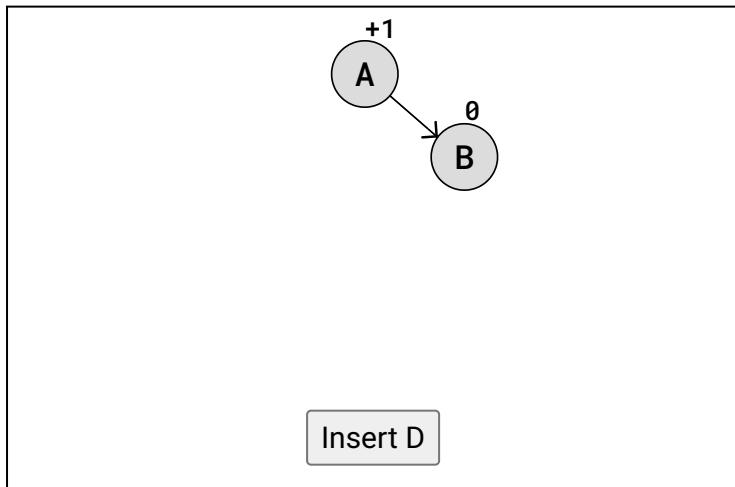
As you step through the animation above, two LL cases happen:

1. When D is added, the balance factor of Q becomes -2, which means the tree is unbalanced. This is an LL case because both the unbalance node Q and its left child node P are left heavy (negative balance factors). A single right rotation at node Q restores the tree balance.
2. After nodes L, C, and B are added, P's balance factor is -2, which means the tree is out of balance. This is also an LL case because both the unbalanced node P and its left child node D are left heavy. A single right rotation restores the balance.

**Note:** The second time the LL case happens in the animation above, a right rotation is done, and L goes from being the right child of D to being the left child of P. Rotations are done like that to keep the correct in-order traversal ('B, C, D, L, P, Q' in the animation above). Another reason for changing parent when a rotation is done is to keep the BST property, that the left child is always lower than the node, and that the right child always higher.

also right heavy.

A single left rotation at the unbalanced node is enough to restore balance in the RR case.



The RR case happens two times in the animation above:

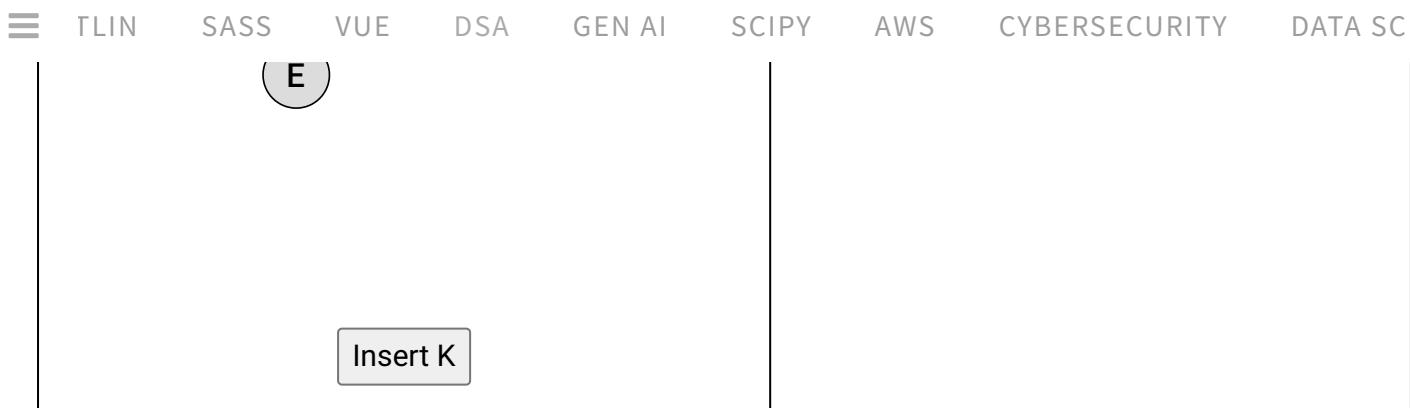
1. When node D is inserted, A becomes unbalanced, and both A and B are right heavy. A left rotation at node A restores the tree balance.
2. After nodes E, C and F are inserted, node B becomes unbalanced. This is an RR case because both node B and its right child node D are right heavy. A left rotation restores the tree balance.

## The Left-Right (LR) Case

The Left-Right case is when the unbalanced node is left heavy, but its left child node is right heavy.

In this LR case, a left rotation is first done on the left child node, and then a right rotation is done on the original unbalanced node.

Step through the animation below to see how the Left-Right case can happen, and how the rotation operations are done to restore balance.



As you are building the AVL Tree in the animation above, the Left-Right case happens 2 times, and rotation operations are required and done to restore balance:

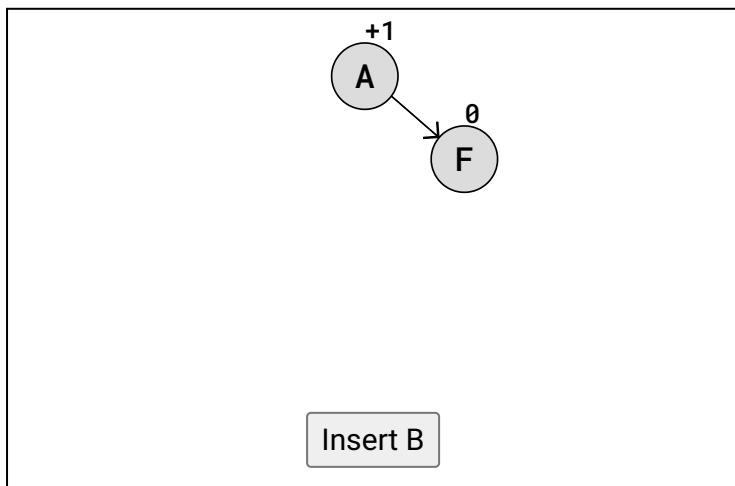
1. When K is inserted, node Q gets unbalanced with a balance factor of -2, so it is left heavy, and its left child E is right heavy, so this is a Left-Right case.
2. After nodes C, F, and G are inserted, node K becomes unbalanced and left heavy, with its left child node E right heavy, so it is a Left-Right case.

## The Right-Left (RL) Case

The Right-Left case is when the unbalanced node is right heavy, and its right child node is left heavy.

In this case we first do a right rotation on the unbalanced node's right child, and then we do a left rotation on the unbalanced node itself.

Step through the animation below to see how the Right-Left case can occur, and how rotations are done to restore the balance.



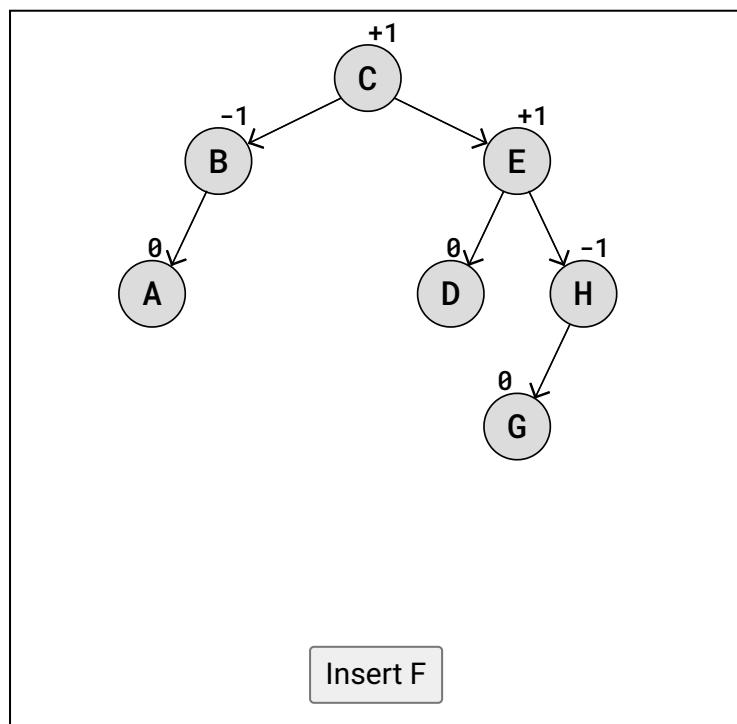
The next Right-Left case occurs after nodes G, E, and D are added. This is a Right-Left case because B is unbalanced and right heavy, and its right child F is left heavy. To restore balance, a right rotation is first done on node F, and then a left rotation is done on node B.

## Retracing in AVL Trees

After inserting or deleting a node in an AVL tree, the tree may become unbalanced. To find out if the tree is unbalanced, we need to update the heights and recalculate the balance factors of all ancestor nodes.

This process, known as retracing, is handled through recursion. As the recursive calls propagate back to the root after an insertion or deletion, each ancestor node's height is updated and the balance factor is recalculated. If any ancestor node is found to have a balance factor outside the range of -1 to 1, a rotation is performed at that node to restore the tree's balance.

In the simulation below, after inserting node F, the nodes C, E and H are all unbalanced, but since retracing works through recursion, the unbalance at node H is discovered and fixed first, which in this case also fixes the unbalance in nodes E and C.



After node F is inserted, the code will retrace, calculating balancing factors as it propagates back up towards the root node. When node H is reached and the balancing factor -2 is calculated, a right rotation is done. Only after this rotation is done, the code will continue to retrace, calculating balancing factors further up on ancestor nodes E and C.

# AVL Insert Node Implementation

This code is based on the BST implementation on the previous page, for inserting nodes.

There is only one new attribute for each node in the AVL tree compared to the BST, and that is the height, but there are many new functions and extra code lines needed for the AVL Tree implementation because of how the AVL Tree rebalances itself.

The implementation below builds an AVL tree based on a list of characters, to create the AVL Tree in the simulation above. The last node to be inserted 'F', also triggers a right rotation, just like in the simulation above.

## Example

Python:

```
class TreeNode:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
        self.height = 1

def getHeight(node):
    if not node:
        return 0
    return node.height

def getBalance(node):
    if not node:
        return 0
    return getHeight(node.left) - getHeight(node.right)

def rightRotate(y):
    print('Rotate right on node', y.data)
    x = y.left
    T2 = x.right
    x.right = y
    y.left = T2
    y.height = 1 + max(getHeight(y.left), getHeight(y.right))
```

```
print('Rotate left on node',x.data)
y = x.right
T2 = y.left
y.left = x
x.right = T2
x.height = 1 + max(getHeight(x.left), getHeight(x.right))
y.height = 1 + max(getHeight(y.left), getHeight(y.right))
return y

def insert(node, data):
    if not node:
        return TreeNode(data)

    if data < node.data:
        node.left = insert(node.left, data)
    elif data > node.data:
        node.right = insert(node.right, data)

    node.height = 1 + max(getHeight(node.left), getHeight(node.right))
    balance = getBalance(node)

    # Left Left
    if balance > 1 and data < node.left.data:
        return rightRotate(node)

    # Right Right
    if balance < -1 and data > node.right.data:
        return leftRotate(node)

    # Left Right
    if balance > 1 and data > node.left.data:
        node.left = leftRotate(node.left)
        return rightRotate(node)

    # Right Left
    if balance < -1 and data < node.right.data:
        node.right = rightRotate(node.right)
        return leftRotate(node)

    return node

def inOrderTraversal(node):
```

```
    inOrderTraversal(node.right)

# Inserting nodes
root = None
letters = ['C', 'B', 'E', 'A', 'D', 'H', 'G', 'F']
for letter in letters:
    root = insert(root, letter)

inOrderTraversal(root)
```

[Run Example »](#)

## AVL Delete Node Implementation

When deleting a node that is not a leaf node, the AVL Tree requires the `minValueNode()` function to find a node's next node in the in-order traversal. This is the same as when deleting a node in a Binary Search Tree, as explained on the previous page.

To delete a node in an AVL Tree, the same code to restore balance is needed as for the code to insert a node.

## Example

Python:

```
def minValueNode(node):
    current = node
    while current.left is not None:
        current = current.left
    return current

def delete(node, data):
    if not node:
        return node

    if data < node.data:
        node.left = delete(node.left, data)
    elif data > node.data:
        node.right = delete(node.right, data)
    else:
        if node.left is None:
            temp = node.right
            node.right = None
            return temp
        elif node.right is None:
            temp = node.left
            node.left = None
            return temp
        temp = minValueNode(node.right)
        node.data = temp.data
        node.right = delete(node.right, temp.data)

    return node
```

```
        temp = node.right
        node = None
        return temp
    elif node.right is None:
        temp = node.left
        node = None
        return temp

    temp = minValueNode(node.right)
    node.data = temp.data
    node.right = delete(node.right, temp.data)

if node is None:
    return node

# Update the balance factor and balance the tree
node.height = 1 + max(getHeight(node.left), getHeight(node.right))
balance = getBalance(node)

# Balancing the tree
# Left Left
if balance > 1 and getBalance(node.left) >= 0:
    return rightRotate(node)

# Left Right
if balance > 1 and getBalance(node.left) < 0:
    node.left = leftRotate(node.left)
    return rightRotate(node)

# Right Right
if balance < -1 and getBalance(node.right) <= 0:
    return leftRotate(node)

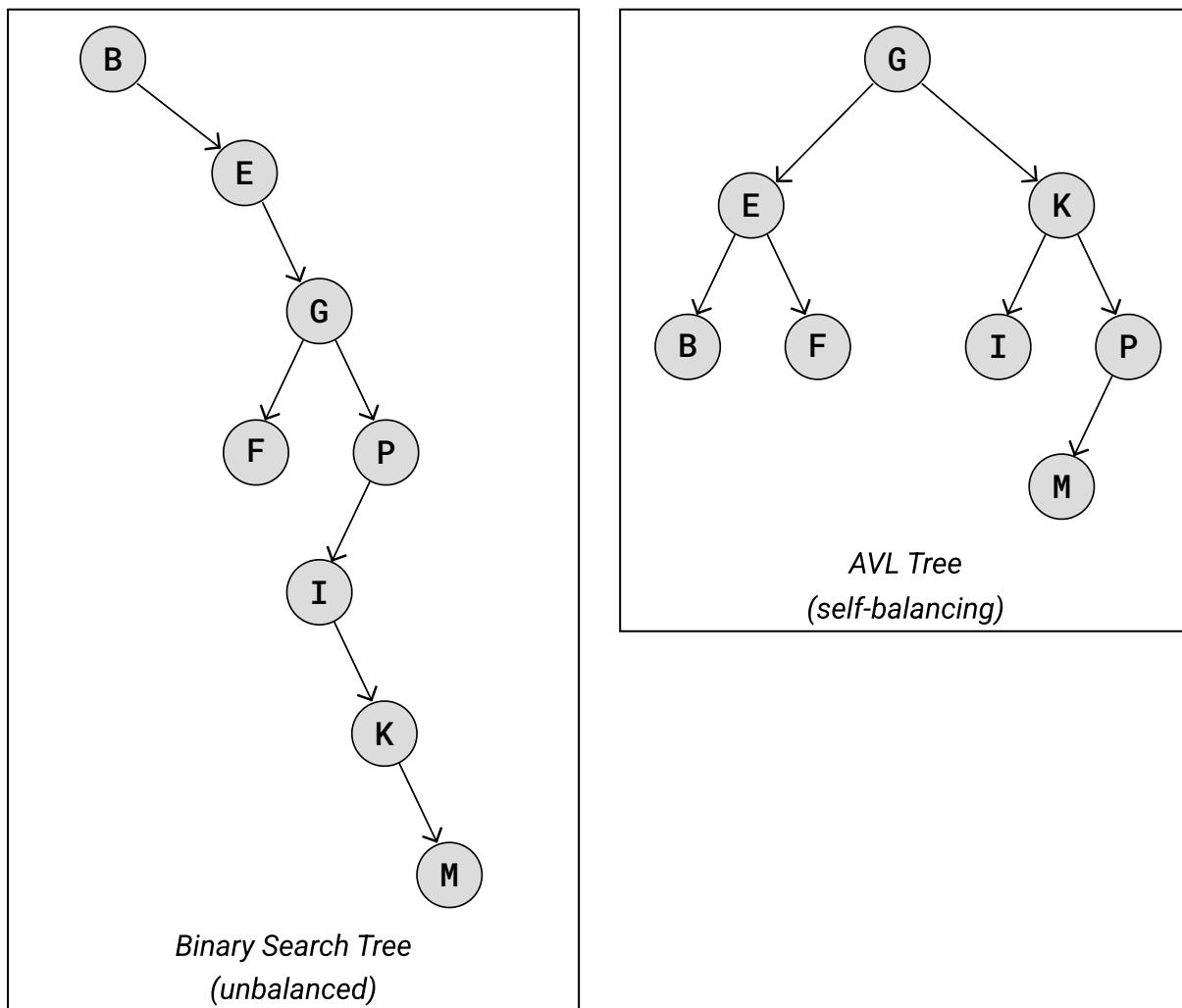
# Right Left
if balance < -1 and getBalance(node.right) > 0:
    node.right = rightRotate(node.right)
    return leftRotate(node)

return node
```

# Time Complexity for AVL Trees

Take a look at the unbalanced Binary Search Tree below. Searching for "M" means that all nodes except 1 must be compared. But searching for "M" in the AVL Tree below only requires us to visit 4 nodes.

So in worst case, algorithms like search, insert, and delete must run through the whole height of the tree. This means that keeping the height ( $h$ ) of the tree low, like we do using AVL Trees, gives us a lower runtime.



See the comparison of the time complexities between Binary Search Trees and AVL Trees below, and how the time complexities relate to the height ( $h$ ) of the tree, and the number of nodes ( $n$ ) in the tree.

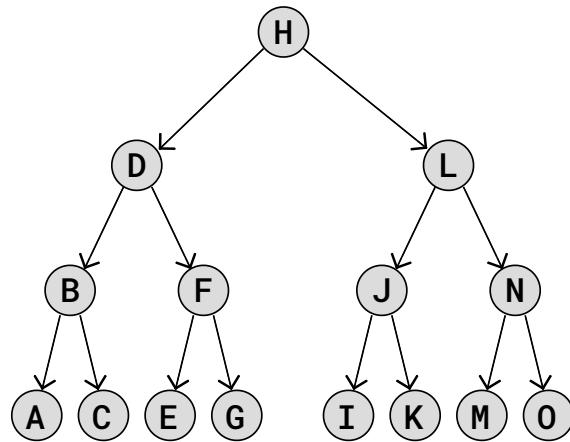
- The **BST** is not self-balancing. This means that a BST can be very unbalanced, almost like a long chain, where the height is nearly the same as the number of nodes. This makes

with time complexity  $O(h) = O(\log n)$ .

# $O(\log n)$ Explained

The fact that the time complexity is  $O(h) = O(\log n)$  for search, insert, and delete on an AVL Tree with height  $h$  and nodes  $n$  can be explained like this:

Imagine a perfect Binary Tree where all nodes have two child nodes except on the lowest level, like the AVL Tree below.



The number of nodes on each level in such an AVL Tree are:

$$1, 2, 4, 8, 16, 32, \dots$$

Which is the same as:

$$2^0, 2^1, 2^2, 2^3, 2^4, 2^5, \dots$$

To get the number of nodes  $n$  in a perfect Binary Tree with height  $h = 3$ , we can add the number of nodes on each level together:

$$n_3 = 2^0 + 2^1 + 2^2 + 2^3 = 15$$

Which is actually the same as:

$$n_3 = 2^4 - 1 = 15$$

And this is actually the case for larger trees as well! If we want to get the number of nodes  $n$  in a tree with height  $h = 5$  for example, we find the number of nodes like this:

$$n_5 = 2^6 - 1 = 63$$

**Note:** The formula above can also be found by calculating the sum of the geometric series

$$2^0 + 2^1 + 2^2 + 2^3 + \dots + 2^n$$

We know that the time complexity for searching, deleting, or inserting a node in an AVL tree is  $O(h)$ , but we want to argue that the time complexity is actually  $O(\log(n))$ , so we need to find the height  $h$  described by the number of nodes  $n$ :

$$\begin{aligned} n &= 2^{h+1} - 1 \\ n + 1 &= 2^{h+1} \\ \log_2(n + 1) &= \log_2(2^{h+1}) \\ h &= \log_2(n + 1) - 1 \\ O(h) &= O(\log n) \end{aligned}$$

How the last line above is derived might not be obvious, but for a Binary Tree with a lot of nodes (big  $n$ ), the "+1" and "-1" terms are not important when we consider time complexity. For more details on how to calculate the time complexity using Big O notation, see [this page](#).

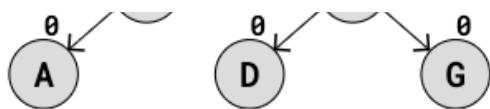
The math above shows that the time complexity for search, delete, and insert operations on an AVL Tree  $O(h)$ , can actually be expressed as  $O(\log n)$ , which is fast, a lot faster than the time complexity for BSTs which is  $O(n)$ .

## DSA Exercises

### Test Yourself With Exercises

#### Exercise:

Each node in the AVL Tree below is displayed together with its balance factor:



What is the balance factor?

The balance factor is the difference between each node's left and right subtree .

[Submit Answer »](#)

[Start the Exercise](#)

[!\[\]\(1a27f322f1e93f8683f14763586f64be\_img.jpg\) Previous](#)

[!\[\]\(2e49d21d7750a050a5cdba609e788680\_img.jpg\) Next](#)

**W3schools Pathfinder**  
Track your progress - it's free!

[Sign Up](#) [Log in](#)

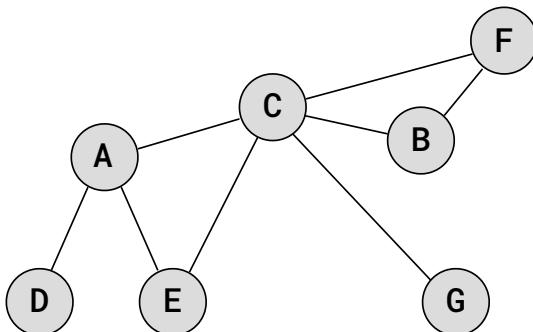
ADVERTISEMENT

# DSA Graphs

[« Previous](#)[Next »](#)

## Graphs

A Graph is a non-linear data structure that consists of vertices (nodes) and edges.



A vertex, also called a node, is a point or an object in the Graph, and an edge is used to connect two vertices with each other.

Graphs are non-linear because the data structure allows us to have different paths to get from one vertex to another, unlike with linear data structures like Arrays or Linked Lists.

Graphs are used to represent and solve problems where the data consists of objects and relationships between them, such as:

- Social Networks: Each person is a vertex, and relationships (like friendships) are the edges. Algorithms can suggest potential friends.
- Maps and Navigation: Locations, like a town or bus stops, are stored as vertices, and roads are stored as edges. Algorithms can find the shortest route between two locations when stored as a Graph.
- Internet: Can be represented as a Graph, with web pages as vertices and hyperlinks as edges.
- Biology: Graphs can model systems like neural networks or the spread of diseases.

Use the animation below to get an understanding of the different Graph properties, and how these properties can be combined.

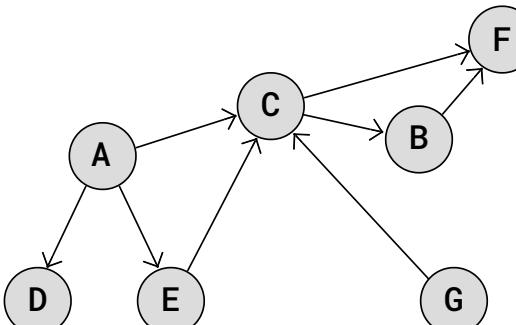
Weighted

Connected

Directed

Cyclic

Loop



```
graph TD; A((A)) --> C((C)); A((A)) --> D((D)); D((D)) --> E((E)); E((E)) --> C((C)); E((E)) --> B((B)); B((B)) --> F((F)); C((C)) --> F((F)); C((C)) --> B((B)); F((F)) --> G((G))
```

A **weighted** Graph is a Graph where the edges have values. The weight value of an edge can represent things like distance, capacity, time, or probability.

A **connected** Graph is when all the vertices are connected through edges somehow. A Graph that is not connected, is a Graph with isolated (disjoint) subgraphs, or single isolated vertices.

A **directed** Graph, also known as a digraph, is when the edges between the vertex pairs have a direction. The direction of an edge can represent things like hierarchy or flow.

A cyclic Graph is defined differently depending on whether it is directed or not:

- A **directed cyclic** Graph is when you can follow a path along the directed edges that goes in circles. Removing the directed edge from F to G in the animation above makes the directed Graph not cyclic anymore.
- An **undirected cyclic** Graph is when you can come back to the same vertex you started at without using the same edge more than once. The undirected Graph above is cyclic because we can start and end up in vertex C without using the same edge twice.

A **loop**, also called a self-loop, is an edge that begins and ends on the same vertex. A loop is a cycle that only consists of one edge. By adding the loop on vertex A in the animation above, the Graph becomes cyclic.

## Graph Representations

A Graph representation tells us how a Graph is stored in memory.

Different Graph representations can:

- be easier to understand and implement than others.

Below are short introductions of the different Graph representations, but Adjacency Matrix is the representation we will use for Graphs moving forward in this tutorial, as it is easy to understand and implement, and works in all cases relevant for this tutorial.

Graph representations store information about which vertices are adjacent, and how the edges between the vertices are. Graph representations are slightly different if the edges are directed or weighted.

Two vertices are adjacent, or neighbors, if there is an edge between them.

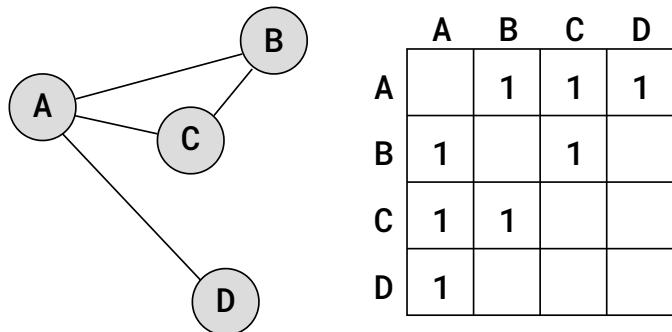
## Adjacency Matrix Graph Representation

Adjacency Matrix is the Graph representation (structure) we will use for this tutorial.

How to implement an Adjacency Matrix is shown on the next page.

The Adjacency Matrix is a 2D array (matrix) where each cell on index  $(i, j)$  stores information about the edge from vertex  $i$  to vertex  $j$ .

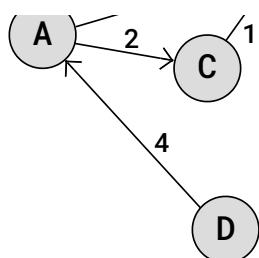
Below is a Graph with the Adjacency Matrix representation next to it.



An undirected Graph  
and the adjacency matrix

The adjacency matrix above represents an undirected Graph, so the values '1' only tells us where the edges are. Also, the values in the adjacency matrix is symmetrical because the edges go both ways (undirected Graph).

To create a directed Graph with an adjacency matrix, we must decide which vertices the edges go from and to, by inserting the value at the correct indexes  $(i, j)$ . To represent a weighted Graph we can put other values than '1' inside the adjacency matrix.



A		3	2	
B				
C		1		
D	4			

*A directed and weighted Graph,  
and its adjacency matrix.*

In the adjacency matrix above, the value **3** on index **(0, 1)** tells us there is an edge from vertex A to vertex B, and the weight for that edge is **3**.

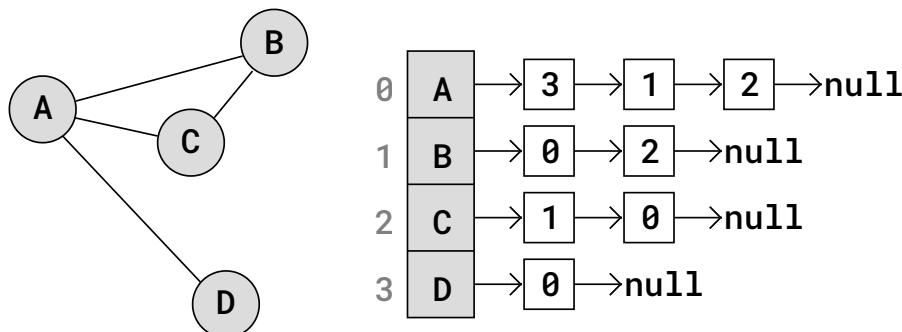
As you can see, the weights are placed directly into the adjacency matrix for the correct edge, and for a directed Graph, the adjacency matrix does not have to be symmetric.

## Adjacency List Graph Representation

In case we have a 'sparse' Graph with many vertices, we can save space by using an Adjacency List compared to using an Adjacency Matrix, because an Adjacency Matrix would reserve a lot of memory on empty Array elements for edges that don't exist.

A 'sparse' Graph is a Graph where each vertex only has edges to a small portion of the other vertices in the Graph.

An Adjacency List has an array that contains all the vertices in the Graph, and each vertex has a Linked List (or Array) with the vertex's edges.

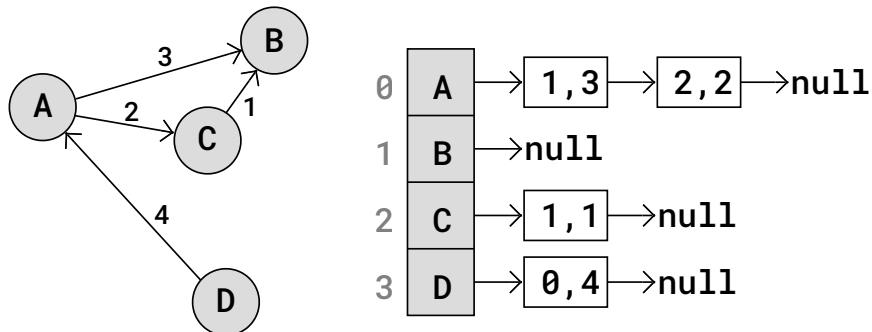


*An undirected Graph  
and its adjacency list.*

In the adjacency list above, the vertices A to D are placed in an Array, and each vertex in the array has its index written right next to it.

indexes to A's adjacent vertices D, B, and C.

An Adjacency List can also represent a directed and weighted Graph, like this:



A directed and weighted Graph  
and its adjacency list.

In the Adjacency List above, vertices are stored in an Array. Each vertex has a pointer to a Linked List with edges stored as `i, w`, where `i` is the index of the vertex the edge goes to, and `w` is the weight of that edge.

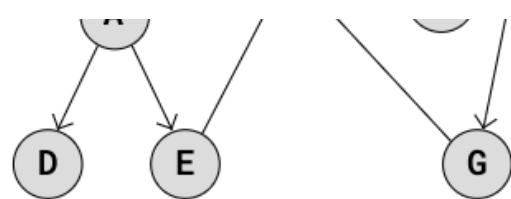
Node D for example, has a pointer to a Linked List with an edge to vertex A. The values `0, 4` means that vertex D has an edge to vertex on index `0` (vertex A), and the weight of that edge is `4`.

## DSA Exercises

## Test Yourself With Exercises

### Exercise:

How can the Graph below be described?



The Graph is cyclic,  
connected, and .

[Submit Answer »](#)

[Start the Exercise](#)

[!\[\]\(f7a44d6bfc33b9779c71539e0ac9d9ba\_img.jpg\) Previous](#)

[!\[\]\(a095e1d9616556409fada8605483d298\_img.jpg\) Next](#)

**W3schools Pathfinder**  
**Track your progress - it's free!**

[Sign Up](#)    [Log in](#)

ADVERTISEMENT

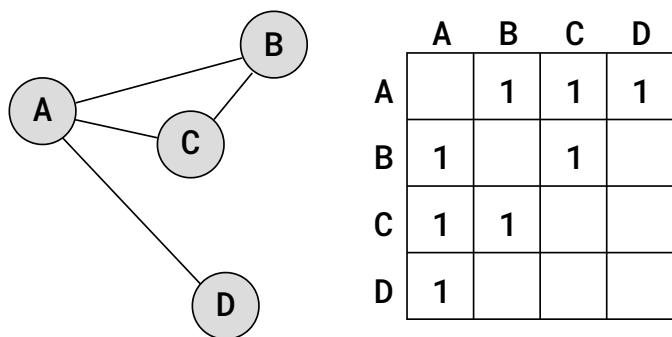
# DSA Graphs Implementation

[« Previous](#)[Next »](#)

## A Basic Graph Implementation

Before we can run algorithms on a Graph, we must first implement it somehow.

To implement a Graph we will use an Adjacency Matrix, like the one below.



An undirected Graph  
and its adjacency matrix

To store data for each vertex, in this case the letters A, B, C, and D, the data is put in a separate array that matches the indexes in the adjacency matrix, like this:

```
vertexData = [ 'A', 'B', 'C', 'D']
```

For an undirected and not weighted Graph, like in the image above, an edge between vertices *i* and *j* is stored with value 1. It is stored as 1 on both places (*j, i*) and (*i, j*) because the

vertex D is on index 3, so we get the edge between A and D stored as value 1 in position (0, 3) and (3, 0), because the edge goes in both directions.

Below is a basic implementation of the undirected Graph from the image above.

## Example

Python:

```
1 vertexData = ['A', 'B', 'C', 'D']
2
3 adjacency_matrix = [
4     [0, 1, 1, 1], # Edges for A
5     [1, 0, 1, 0], # Edges for B
6     [1, 1, 0, 0], # Edges for C
7     [1, 0, 0, 0]  # Edges for D
8 ]
9
10 def print_adjacency_matrix(matrix):
11     print("\nAdjacency Matrix:")
12     for row in matrix:
13         print(row)
14
15 print('vertexData:', vertexData)
16 print_adjacency_matrix(adjacency_matrix)
```

[Run Example »](#)

This implementation is basically just a two dimensional array, but to get a better sense of how the vertices are connected by edges in the Graph we have just implemented, we can run this function:

## Example

Python:

```
1 def print_connections(matrix, vertices):
2     print("\nConnections for each vertex:")
3
```

≡ TLIN

SASS

VUE

DSA

GEN AI

SCIPY

AWS

CYBERSECURITY

DATA SC

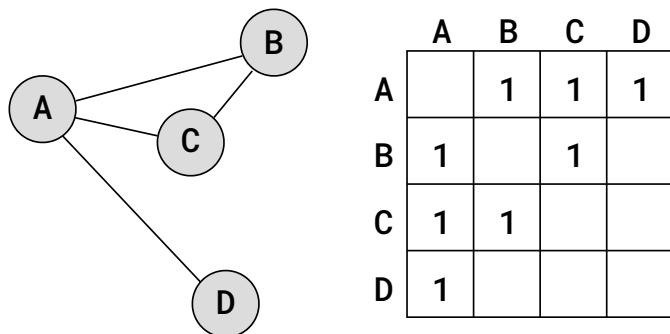
```
' 8
      print(vertices[j], end=" ")
print() # new line
```

[Run Example »](#)

## Graph Implementation Using Classes

A more proper way to store a Graph is to add an abstraction layer using classes so that a Graph's vertices, edges, and relevant methods, like algorithms that we will implement later, are contained in one place.

Programming languages with built-in object-oriented functionality like Python and Java, make implementation of Graphs using classes much easier than languages like C, without this built-in functionality.



*An undirected Graph  
and its adjacency matrix*

Here is how the undirected Graph above can be implemented using classes.

## Example

Python:

```
1 class Graph:
2     def __init__(self, size):
3         self.adj_matrix = [[0] * size for _ in range(size)]
4         self.size = size
5         self.vertex_data = [''] * size
```

```
11
12     def add_vertex_data(self, vertex, data):
13         if 0 <= vertex < self.size:
14             self.vertex_data[vertex] = data
15
16     def print_graph(self):
17         print("Adjacency Matrix:")
18         for row in self.adj_matrix:
19             print(' '.join(map(str, row)))
20         print("\nVertex Data:")
21         for vertex, data in enumerate(self.vertex_data):
22             print(f"Vertex {vertex}: {data}")
23
24 g = Graph(4)
25 g.add_vertex_data(0, 'A')
26 g.add_vertex_data(1, 'B')
27 g.add_vertex_data(2, 'C')
28 g.add_vertex_data(3, 'D')
29
30
31
32
33
34 g.print_graph()
```

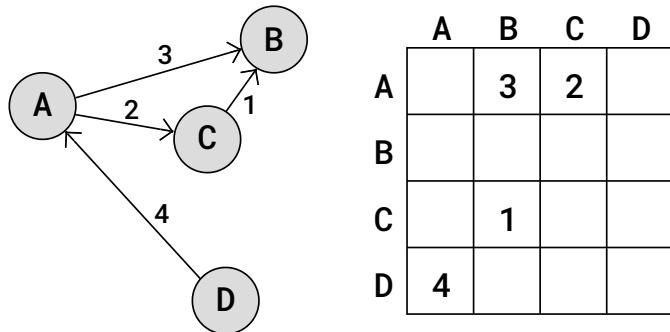
[Run Example »](#)

In the code above, the matrix symmetry we get for undirected Graphs is provided for on line 9 and 10, and this saves us some code when initializing the edges in the Graph on lines 29-32.

## Implementation of Directed and Weighted Graphs

To implement a Graph that is directed and weighted, we just need to do a few changes to previous implementation of the undirected Graph.

so that instead of just having value 1 to indicate that there is an edge between two vertices, we use the actual weight value to define the edge.



*A directed and weighted Graph,  
and its adjacency matrix.*

Below is the implementation of the directed and weighted Graph above.

## Example

Python:

```
class Graph:
    def __init__(self, size):
        self.adj_matrix = [[None] * size for _ in range(size)]
        self.size = size
        self.vertex_data = [''] * size

    def add_edge(self, u, v, weight):
        if 0 <= u < self.size and 0 <= v < self.size:
            self.adj_matrix[u][v] = weight
            self.adj_matrix[v][u] = weight

    def add_vertex_data(self, vertex, data):
        if 0 <= vertex < self.size:
            self.vertex_data[vertex] = data

    def print_graph(self):
        print("Adjacency Matrix:")
        for row in self.adj_matrix:
            print(' '.join(map(lambda x: str(x) if x is not None else 'None', row)))
        print("\nVertex Data:")
        for vertex, data in enumerate(self.vertex_data):
```

```
26 g.add_vertex_data(1, 'B')
27 g.add_vertex_data(2, 'C')
28 g.add_vertex_data(3, 'D')
29 g.add_edge(0, 1, 3) # A -> B with weight 3
30 g.add_edge(0, 2, 2) # A -> C with weight 2
31 g.add_edge(3, 0, 4) # D -> A with weight 4
32 g.add_edge(2, 1, 1) # C -> B with weight 1
33
34 g.print_graph()
```

[Run Example »](#)

**Line 3:** All edges are set to `None` initially.

**Line 7:** The weight can now be added to an edge with the additional `weight` argument.

**Line 10:** By removing line 10, the Graph can now be set up as being directed.

On the next page we will see how Graphs can be traversed, and on the next pages after that we will look at different algorithms that can run on the Graph data structure.

## DSA Exercises

### Test Yourself With Exercises

#### Exercise:

How are the edges in a graph implemented?

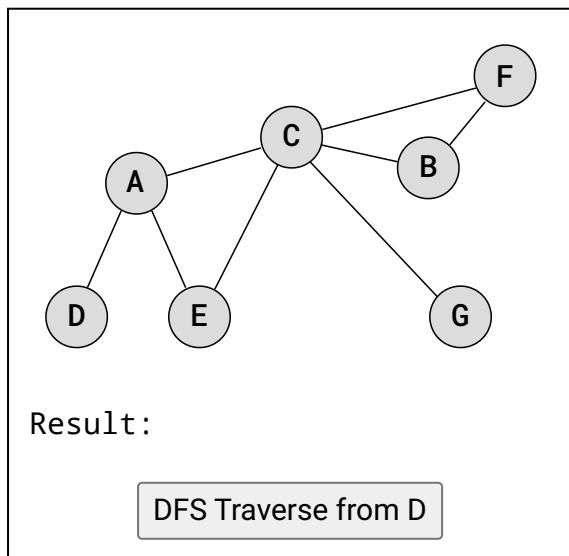
The edges, and edge weights,  
in a graph are normally

# DSA Graphs Traversal

[« Previous](#)[Next »](#)

## Graphs Traversal

To traverse a Graph means to start in one vertex, and go along the edges to visit other vertices until all vertices, or as many as possible, have been visited.



Understanding how a Graph can be traversed is important for understanding how algorithms that run on Graphs work.

The two most common ways a Graph can be traversed are:

- Depth First Search (DFS)
- Breadth First Search (BFS)

DFS is usually implemented using a Stack or by the use of recursion (which utilizes the call stack), while BFS is usually implemented using a Queue.

running. Once FunctionB is finished, it is removed from the stack, and then FunctionA resumes its work.

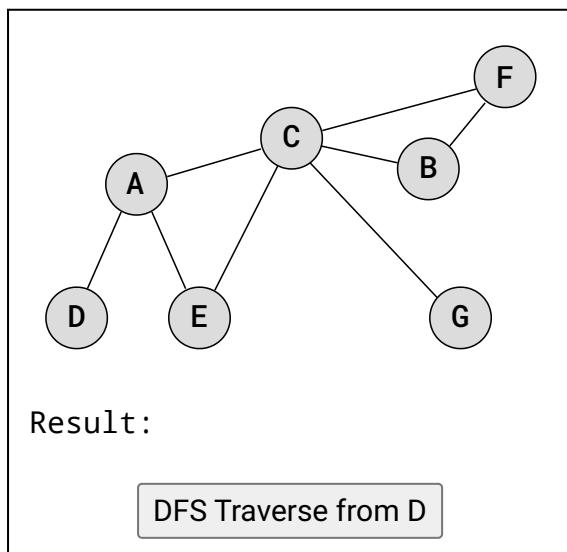
## Depth First Search Traversal

Depth First Search is said to go "deep" because it visits a vertex, then an adjacent vertex, and then that vertex' adjacent vertex, and so on, and in this way the distance from the starting vertex increases for each recursive iteration.

### How it works:

1. Start DFS traversal on a vertex.
2. Do a recursive DFS traversal on each of the adjacent vertices as long as they are not already visited.

Run the animation below to see how Depth First Search (DFS) traversal runs on a specific Graph, starting in vertex D (it is the same as the previous animation).



The DFS traversal starts in vertex D, marks vertex D as visited. Then, for every new vertex visited, the traversal method is called recursively on all adjacent vertices that have not been visited yet. So

## Example

Python:

```
1  class Graph:
2      def __init__(self, size):
3          self.adj_matrix = [[0] * size for _ in range(size)]
4          self.size = size
5          self.vertex_data = [''] * size
6
7      def add_edge(self, u, v):
8          if 0 <= u < self.size and 0 <= v < self.size:
9              self.adj_matrix[u][v] = 1
10             self.adj_matrix[v][u] = 1
11
12     def add_vertex_data(self, vertex, data):
13         if 0 <= vertex < self.size:
14             self.vertex_data[vertex] = data
15
16     def print_graph(self):
17         print("Adjacency Matrix:")
18         for row in self.adj_matrix:
19             print(' '.join(map(str, row)))
20         print("\nVertex Data:")
21         for vertex, data in enumerate(self.vertex_data):
22             print(f"Vertex {vertex}: {data}")
23
```

36

37 g = Graph(7)

38

```
43 g.add_vertex_data(4, 'E')
44 g.add_vertex_data(5, 'F')
45 g.add_vertex_data(6, 'G')
46
47 g.add_edge(3, 0) # D - A
48 g.add_edge(0, 2) # A - C
49 g.add_edge(0, 3) # A - D
50 g.add_edge(0, 4) # A - E
51 g.add_edge(4, 2) # E - C
52 g.add_edge(2, 5) # C - F
53 g.add_edge(2, 1) # C - B
54 g.add_edge(2, 6) # C - G
55 g.add_edge(1, 5) # B - F
56
57 g.print_graph()
58
59 print("\nDepth First Search starting from vertex D:")
```

[Run Example »](#)

**Line 60:** The DFS traversal starts when the `dfs()` method is called.

**Line 33:** The `visited` array is first set to `false` for all vertices, because no vertices are visited yet at this point.

**Line 35:** The `visited` array is sent as an argument to the `dfs_util()` method. When the `visited` array is sent as an argument like this, it is actually just a reference to the `visited` array that is sent to the `dfs_util()` method, and not the actual array with the values inside. So there is always just one `visited` array in our program, and the `dfs_util()` method can make changes to it as nodes are visited (line 25).

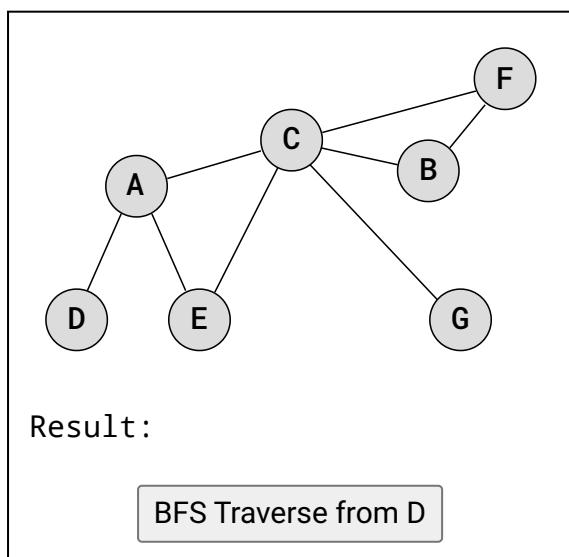
**Line 28-30:** For the current vertex `v`, all adjacent nodes are called recursively if they are not already visited.

## Breadth First Search Traversal

## How it works:

1. Put the starting vertex into the queue.
2. For each vertex taken from the queue, visit the vertex, then put all unvisited adjacent vertices into the queue.
3. Continue as long as there are vertices in the queue.

Run the animation below to see how Breadth First Search (BFS) traversal runs on a specific Graph, starting in vertex D.



As you can see in the animation above, BFS traversal visits vertices the same distance from the starting vertex, before visiting vertices further away. So for example, after visiting vertex A, vertex E and C are visited before visiting B, F and G because those vertices are further away.

Breadth First Search traversal works this way by putting all adjacent vertices in a queue (if they are not already visited), and then using the queue to visit the next vertex.

This code example for Breadth First Search traversal is the same as for the Depth First Search code example above, except for the `bfs()` method:

## Example

Python:

```
TLIN SASS VUE DSA GEN AI SCIPY AWS CYBERSECURITY DATA SC
4     visited[queue[0]] = True
5
6     while queue:
7         current_vertex = queue.pop(0)
8         print(self.vertex_data[current_vertex], end=' ')
9
10        for i in range(self.size):
11            if self.adj_matrix[current_vertex][i] == 1 and not vis
12                queue.append(i)
13                visited[i] = True
```

[Run Example »](#)

**Line 2-4:** The `bfs()` method starts by creating a queue with the start vertex inside, creating a `visited` array, and setting the start vertex as visited.

**Line 6-13:** The BFS traversal works by taking a vertex from the queue, printing it, and adding adjacent vertices to the queue if they are not visited yet, and then continue to take vertices from the queue in this way. The traversal finishes when the last element in the queue has no unvisited adjacent vertices.

## DFS and BFS Traversal of a Directed Graph

Depth first and breadth first traversals can actually be implemented to work on directed Graphs (instead of undirected) with just very few changes.

Run the animation below to see how a directed Graph can be traversed using DFS or BFS.

TLIN SASS VUE DSA GEN AI SCIPY AWS CYBERSECURITY DATA SC

Result:

BFS  
 DFS

Traverse from D

To go from traversing a directed Graph instead of an undirected Graph, we just need to remove the last line in the `add_edge()` method:

```
def add_edge(self, u, v):
    if 0 <= u < self.size and 0 <= v < self.size:
        self.adj_matrix[u][v] = 1
        self.adj_matrix[v][u] = 1
```

We must also take care when we build our Graph because the edges are now directed.

The code example below contains both BFS and DFS traversal of the directed Graph from the animation above:

## Example

Python:

```
class Graph:
    def __init__(self, size):
        self.adj_matrix = [[0] * size for _ in range(size)]
        self.size = size
        self.vertex_data = [''] * size

    def add_edge(self, u, v):
```

```
def add_vertex_data(self, vertex, data):
    if 0 <= vertex < self.size:
        self.vertex_data[vertex] = data

def print_graph(self):
    print("Adjacency Matrix:")
    for row in self.adj_matrix:
        print(' '.join(map(str, row)))
    print("\nVertex Data:")
    for vertex, data in enumerate(self.vertex_data):
        print(f"Vertex {vertex}: {data}")

def dfs_util(self, v, visited):
    visited[v] = True
    print(self.vertex_data[v], end=' ')
    for i in range(self.size):
        if self.adj_matrix[v][i] == 1 and not visited[i]:
            self.dfs_util(i, visited)

def dfs(self, start_vertex_data):
    visited = [False] * self.size
    start_vertex = self.vertex_data.index(start_vertex_data)
    self.dfs_util(start_vertex, visited)

def bfs(self, start_vertex_data):
    queue = [self.vertex_data.index(start_vertex_data)]
    visited = [False] * self.size
    visited[queue[0]] = True

    while queue:
        current_vertex = queue.pop(0)
        print(self.vertex_data[current_vertex], end=' ')
        for i in range(self.size):
            if self.adj_matrix[current_vertex][i] == 1 and not visited[i]:
                queue.append(i)
                visited[i] = True

g = Graph(7)
```

```
g.add_vertex_data(4, 'E')
g.add_vertex_data(5, 'F')
g.add_vertex_data(6, 'G')

g.add_edge(3, 0) # D -> A
g.add_edge(3, 4) # D -> E
g.add_edge(4, 0) # E -> A
g.add_edge(0, 2) # A -> C
g.add_edge(2, 5) # C -> F
g.add_edge(2, 6) # C -> G
g.add_edge(5, 1) # F -> B
g.add_edge(1, 2) # B -> C

g.print_graph()

print("\nDepth First Search starting from vertex D:")
g.dfs('D')

print("\n\nBreadth First Search starting from vertex D:")
g.bfs('D')
```

[Run Example »](#)

Now that we have looked at two basic algorithms for how to traverse Graphs, we will use the next pages to see how other algorithms can run on the Graph data structure.

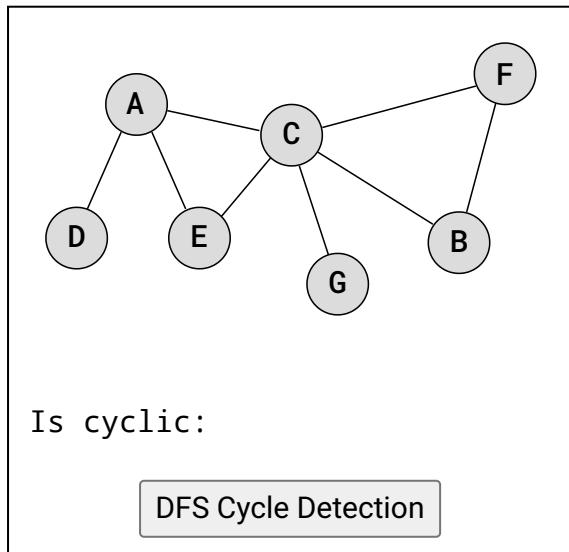
[« Previous](#)[Next »](#)**W3schools Pathfinder****Track your progress - it's free!**[Sign Up](#)[Log in](#)

# DSA Graphs Cycle Detection

[« Previous](#)[Next »](#)

## Cycles in Graphs

A cycle in a Graph is a path that starts and ends at the same vertex, where no edges are repeated. It is similar to walking through a maze and ending up exactly where you started.



A cycle can be defined slightly different depending on the situation. A self-loop for example, where an edge goes from and to the same vertex, might or might not be considered a cycle, depending on the problem you are trying to solve.

## Cycle Detection

It is important to be able to detect cycles in Graphs because cycles can indicate problems or special conditions in many applications like networking, scheduling, and circuit design.

visited.

2. **Union-Find:** This works by initially defining each vertex as a group, or a subset. Then these groups are joined for every edge. Whenever a new edge is explored, a cycle is detected if two vertices already belong to the same group.

How cycle detection with DFS and Union-Find work, and how they are implemented, are explained in more detail below.

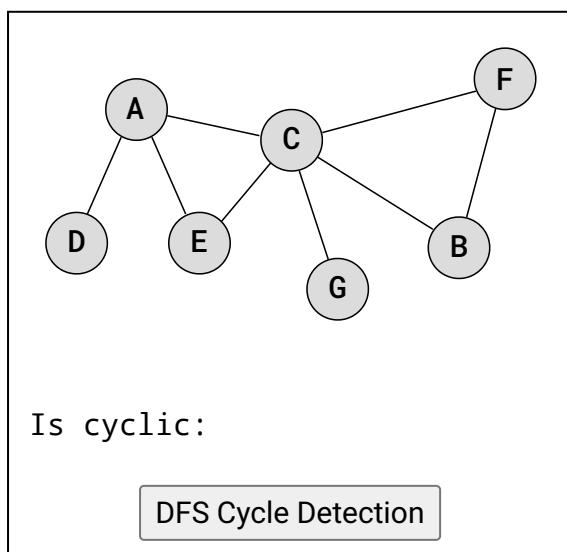
## DFS Cycle Detection for Undirected Graphs

To detect cycles in an undirected Graph using Depth First Search (DFS), we use a code very similar to [the DFS traversal code](#) on the previous page, with just a few changes.

### How it works:

1. Start DFS traversal on each unvisited vertex (in case the Graph is not connected).
2. During DFS, mark vertices as visited, and run DFS on the adjacent vertices (recursively).
3. If an adjacent vertex is already visited and is not the parent of the current vertex, a cycle is detected, and `True` is returned.
4. If DFS traversal is done on all vertices and no cycles are detected, `False` is returned.

Run the animation below to see how DFS cycle detection runs on a specific Graph, starting in vertex A (this is the same as the previous animation).



## Example

Python:

```
1  class Graph:
2      def __init__(self, size):
3          self.adj_matrix = [[0] * size for _ in range(size)]
4          self.size = size
5          self.vertex_data = [''] * size
6
7      def add_edge(self, u, v):
8          if 0 <= u < self.size and 0 <= v < self.size:
9              self.adj_matrix[u][v] = 1
10             self.adj_matrix[v][u] = 1
11
12     def add_vertex_data(self, vertex, data):
13         if 0 <= vertex < self.size:
14             self.vertex_data[vertex] = data
15
16     def print_graph(self):
17         print("Adjacency Matrix:")
18         for row in self.adj_matrix:
19             print(' '.join(map(str, row)))
20         print("\nVertex Data:")
21         for vertex, data in enumerate(self.vertex_data):
22             print(f"Vertex {vertex}: {data}")
23
```

```
43
44 g = Graph(7)
45
46 g.add_vertex_data(0, 'A')
47 g.add_vertex_data(1, 'B')
48 g.add_vertex_data(2, 'C')
49 g.add_vertex_data(3, 'D')
50 g.add_vertex_data(4, 'E')
51 g.add_vertex_data(5, 'F')
52 g.add_vertex_data(6, 'G')
53
54 g.add_edge(3, 0) # D - A
55 g.add_edge(0, 2) # A - C
56 g.add_edge(0, 3) # A - D
57 g.add_edge(0, 4) # A - E
58 g.add_edge(4, 2) # E - C
59 g.add_edge(2, 5) # C - F
60 g.add_edge(2, 1) # C - B
61 g.add_edge(2, 6) # C - G
62 g.add_edge(1, 5) # B - F
63
64 g.print_graph()
65
```

[Run Example »](#)

**Line 66:** The DFS cycle detection starts when the `is_cyclic()` method is called.

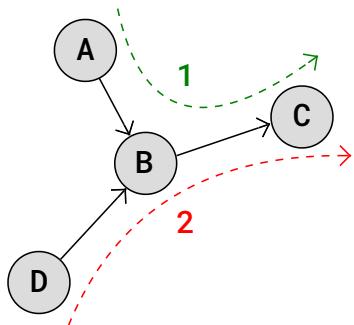
**Line 37:** The `visited` array is first set to `false` for all vertices, because no vertices are visited yet at this point.

**Line 38-42:** DFS cycle detection is run on all vertices in the Graph. This is to make sure all vertices are visited in case the Graph is not connected. If a node is already visited, there must be a cycle, and `True` is returned. If all nodes are visited just ones, which means no cycles are detected, `False` is returned.

# DFS Cycle Detection for Directed Graphs

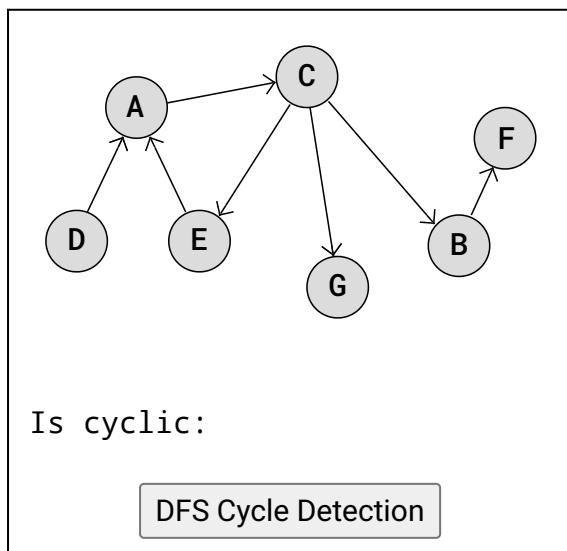
To detect cycles in Graphs that are directed, the algorithm is still very similar as for undirected Graphs, but the code must be modified a little bit because for a directed Graph, if we come to an adjacent node that has already been visited, it does not necessarily mean that there is a cycle.

Just consider the following Graph where two paths are explored, trying to detect a cycle:



In path 1, the first path to be explored, vertices A->B->C are visited, no cycles detected.

In the second path to be explored (path 2), vertices D->B->C are visited, and the path has no cycles, right? But without changes in our program, a false cycle would actually be detected when going from D to the adjacent vertex B, because B has already been visited in path 1. To avoid such false detections, the code is modified to detect cycles only in case a node has been visited before in the same path.



## Example

Python:

```
1  class Graph:
2      # .....
3      def add_edge(self, u, v):
4          if 0 <= u < self.size and 0 <= v < self.size:
5              self.adj_matrix[u][v] = 1
6
7      # .....
8      def dfs_util(self, v, visited, recStack):
9          visited[v] = True
10         recStack[v] = True
11         print("Current vertex:",self.vertex_data[v])
12
13         for i in range(self.size):
14
15             if self.adj_matrix[v][i] == 1 and not visited[i]:
16                 if recStack[i]:
17                     return True
18
19             recStack[i] = False
20             return False
21
22
23
24     def is_cyclic(self):
25         visited = [False] * self.size
26
27         for i in range(self.size):
28             if not visited[i]:
29                 print() #new line
30                 if self.dfs_util(i, visited, recStack):
31                     return True
32
33
34 g = Graph(7)
35
36 # .....
```

```
42 | g.add_edge(2, 4) # C -> E
43 | g.add_edge(1, 5) # B -> F
44 | g.add_edge(4, 0) # E -> A
45 | g.add_edge(2, 6) # C -> G
46 |
47 | g.print_graph()
48 |
      print("Graph has cycle:", g.is_cyclic())
```

[Run Example »](#)

**Line 6:** This line is removed because it is only applicable for undirected Graphs.

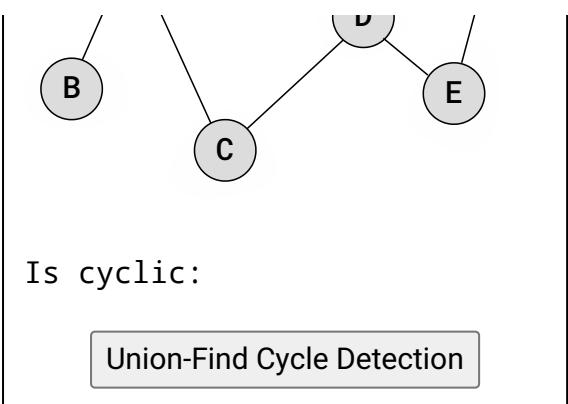
**Line 26:** The `recStack` array keeps an overview over which vertices have been visited during a recursive exploration of a path.

**Line 14-19:** For every adjacent vertex not visited before, do a recursive DFS cycle detection. If an adjacent vertex has been visited before, also in the same recursive path (line 13), a cycle has been found, and `True` is returned.

## Union-Find Cycle Detection

Detecting cycles using Union-Find is very different from using Depth First Search.

Union-Find cycle detection works by first putting each node in its own subset (like a bag or container). Then, for every edge, the subsets belonging to each vertex are merged. For an edge, if the vertices already belong to the same subset, it means that we have found a cycle.



In the animation above, Union-Find cycle detection explores the edges in the Graph. As edges are explored, the subset of vertex A grows to also include vertices B, C, and D. The cycle is detected when the edge between A and D is explored, and it is discovered that both A and D already belong to the same subset.

The edges between D, E, and F also construct a circle, but this circle is not detected because the algorithm stops (returns `True`) when the first circle is detected.

Union-Find cycle detection is only applicable for Graphs that are undirected.

Union-Find cycle detection is implemented using the [adjacency matrix representation](#), so setting up the Graph structure with vertices and edges is basically the same as in previous examples.

## Example

Python:

```
class Graph:
    def __init__(self, size):
        self.adj_matrix = [[0] * size for _ in range(size)]
        self.size = size
        self.vertex_data = [''] * size
        self.parent = [i for i in range(size)] # Union-Find array

    def add_edge(self, u, v):
        if 0 <= u < self.size and 0 <= v < self.size:
            self.adj_matrix[u][v] = 1
            self.adj_matrix[v][u] = 1

    def add_vertex_data(self, vertex, data):
        if 0 <= vertex < self.size:
```

```
39 |         return True
40 |
41 | g = Graph(7)
42 |
43 | g.add_vertex_data(0, 'A')
44 | g.add_vertex_data(1, 'B')
45 | g.add_vertex_data(2, 'C')
46 | g.add_vertex_data(3, 'D')
47 | g.add_vertex_data(4, 'E')
48 | g.add_vertex_data(5, 'F')
49 | g.add_vertex_data(6, 'G')
50 |
51 | g.add_edge(1, 0)    # B - A
52 | g.add_edge(0, 3)    # A - D
53 | g.add_edge(0, 2)    # A - C
54 | g.add_edge(2, 3)    # C - D
55 | g.add_edge(3, 4)    # D - E
56 | g.add_edge(3, 5)    # D - F
57 | g.add_edge(3, 6)    # D - G
58 | g.add_edge(4, 5)    # E - F
59 |
60 | print("Graph has cycle:", g.is_cyclic())
```

**Line 6:** The `parent` array contains the root vertex for every subset. This is used to detect a cycle by checking if two vertices on either side of an edge already belong to the same subset.

**Line 17:** The `find` method finds the root of the set that the given vertex belongs to.

**Line 22:** The `union` method combines two subsets.

**Line 29:** The `is_cyclic` method uses the `find` method to detect a cycle if two vertices `x` and `y` are already in the same subset. If a cycle is not detected, the `union` method is used to combine the subsets.

## DSA Exercises

### Test Yourself With Exercises

#### Exercise:

What is a cycle in a Graph?

A cycle in a Graph is a path  
that starts and ends at the  
same , where no  
are repeated.

[Submit Answer »](#)

[Start the Exercise](#)

# DSA Shortest Path

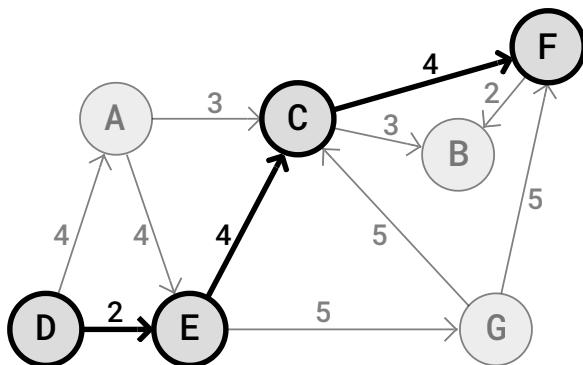
[« Previous](#)[Next »](#)

## The Shortest Path Problem

The shortest path problem is famous in the field of computer science.

To solve the shortest path problem means to find the shortest possible route or path between two vertices (or nodes) in a Graph.

In the shortest path problem, a Graph can represent anything from a road network to a communication network, where the vertices can be intersections, cities, or routers, and the edges can be roads, flight paths, or data links.



The shortest path from vertex D to vertex F in the Graph above is D->E->C->F, with a total path weight of  $2+4+4=10$ . Other paths from D to F are also possible, but they have a higher total weight, so they can not be considered to be the shortest path.

## Solutions to The Shortest Path Problem

where we can move from one vertex to another using the lowest possible combined weight along the edges.

This sum of weights along the edges that make up a path is called a **path cost** or a **path weight**.

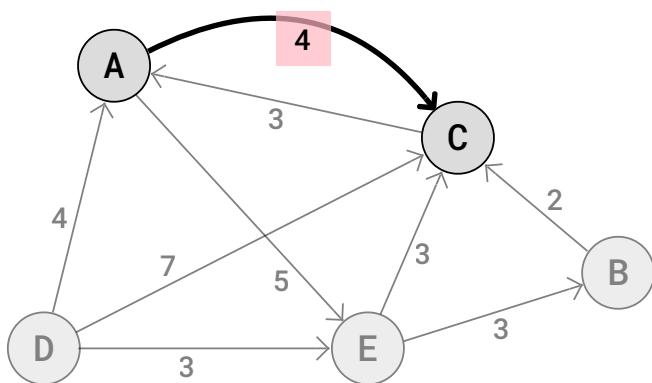
Algorithms that find the shortest paths, like [Dijkstra's algorithm](#) or [the Bellman-Ford algorithm](#), find the shortest paths from one start vertex to all other vertices.

To begin with, the algorithms set the distance from the start vertex to all vertices to be infinitely long. And as the algorithms run, edges between the vertices are checked over and over, and shorter paths might be found many times until the shortest paths are found at the end.

Every time an edge is checked and it leads to a shorter distance to a vertex being found and updated, it is called a **relaxation**, or **relaxing** an edge.

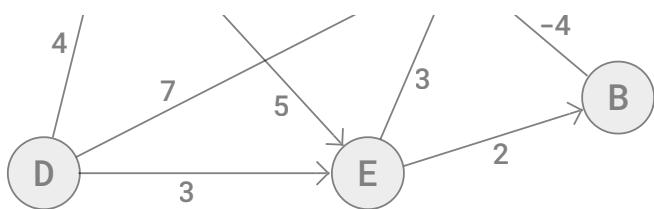
## Positive and Negative Edge Weights

Some algorithms that find the shortest paths, like [Dijkstra's algorithm](#), can only find the shortest paths in graphs where all the edges are positive. Such graphs with positive distances are also the easiest to understand because we can think of the edges between vertices as distances between locations.



If we interpret the edge weights as money lost by going from one vertex to another, a positive edge weight of 4 from vertex A to C in the graph above means that we must spend \$4 to go from A to C.

But graphs can also have negative edges, and for such graphs [the Bellman-Ford algorithm](#) can be used to find the shortest paths.



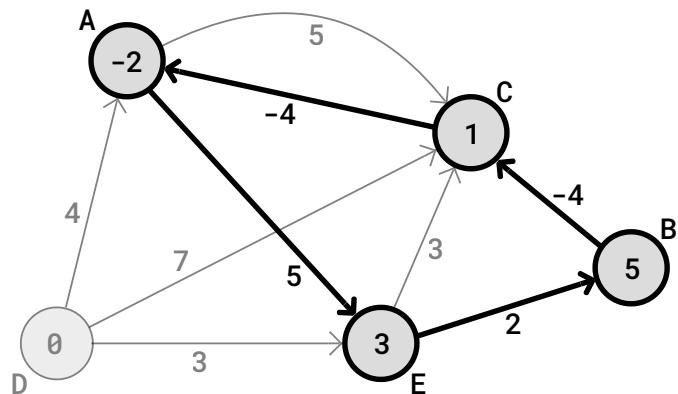
And similarly, if the edge weights represent money lost, the negative edge weight -3 from vertex C to A in the graph above can be understood as an edge where there is more money to be made than money lost by going from C to A. So if for example the cost of fuel is \$5 going from C to A, and we get paid \$8 for picking up packages in C and delivering them in A, money lost is -3, meaning we are actually earning \$3 in total.

## Negative Cycles in Shortest Path Problems

Finding the shortest paths becomes impossible if a graph has negative cycles.

Having a negative cycle means that there is a path where you can go in circles, and the edges that make up this circle have a total path weight that is negative.

In the graph below, the path A->E->B->C->A is a negative cycle because the total path weight is  $5+2-4-4=-1$ .



The reason why it is impossible to find the shortest paths in a graph with negative cycles is that it will always be possible to continue running an algorithm to find even shorter paths.

Let's say for example that we are looking for the shortest distance from vertex D in graph above, to all other vertices. At first we find the distance from D to E to be 3, by just walking the edge D->E. But after this, if we walk one round in the negative cycle E->B->C->A->E, then the distance to E becomes 2. After walking one more round the distance becomes 1, which is even shorter, and so

# DSA Dijkstra's Algorithm

[« Previous](#)[Next »](#)

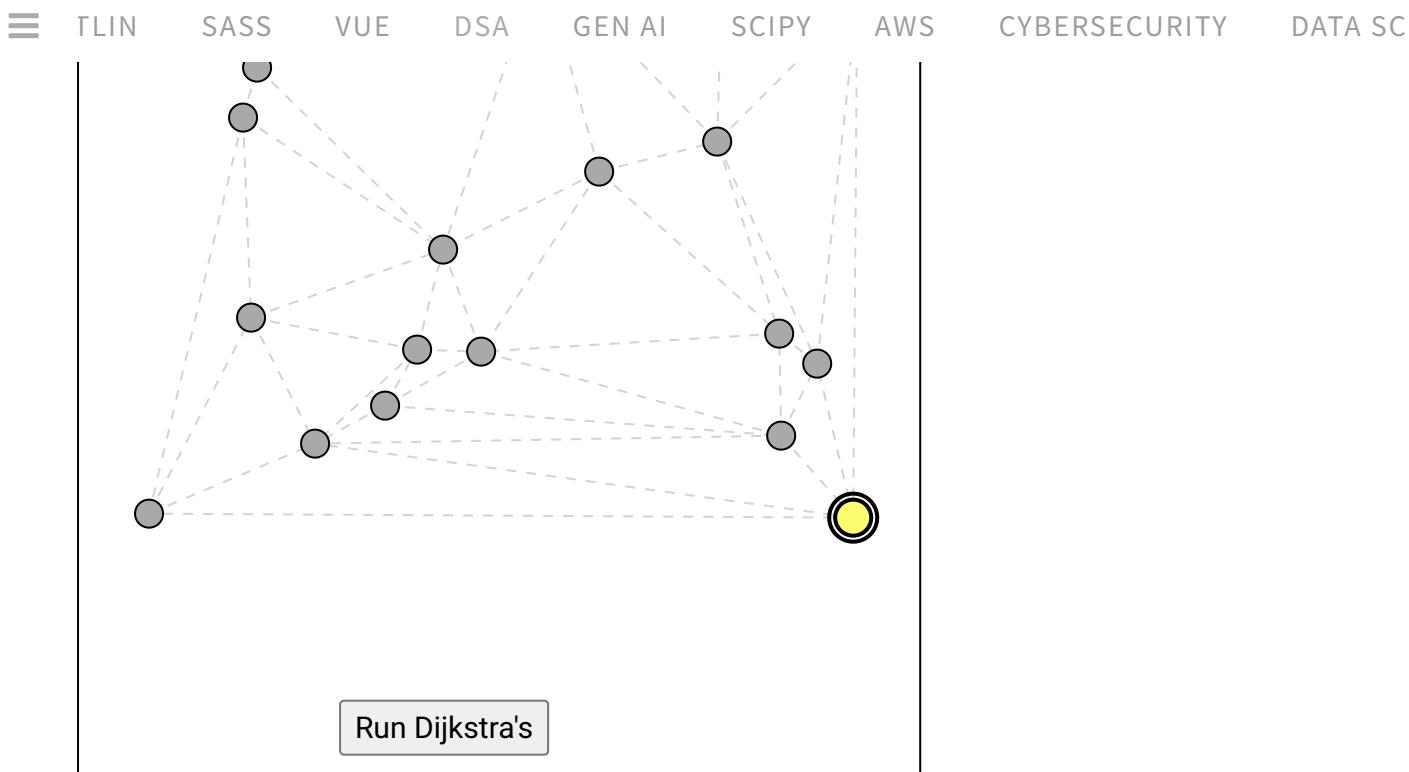
Dijkstra's shortest path algorithm was invented in 1956 by the Dutch computer scientist Edsger W. Dijkstra during a twenty minutes coffee break, while out shopping with his fiancée in Amsterdam.

The reason for inventing the algorithm was to test a new computer called ARMAC.

## Dijkstra's Algorithm

Dijkstra's algorithm finds the shortest path from one vertex to all other vertices.

It does so by repeatedly selecting the nearest unvisited vertex and calculating the distance to all the unvisited neighboring vertices.



Dijkstra's algorithm is often considered to be the most straightforward algorithm for solving the shortest path problem.

Dijkstra's algorithm is used for solving single-source shortest path problems for directed or undirected graphs. Single-source means that one vertex is chosen to be the start, and the algorithm will find the shortest path from that vertex to all other vertices.

Dijkstra's algorithm does not work for graphs with negative edges. For graphs with negative edges, the Bellman-Ford algorithm that is described on the next page, can be used instead.

To find the shortest path, Dijkstra's algorithm needs to know which vertex is the source, it needs a way to mark vertices as visited, and it needs an overview of the current shortest distance to each vertex as it works its way through the graph, updating these distances when a shorter distance is found.

### How it works:

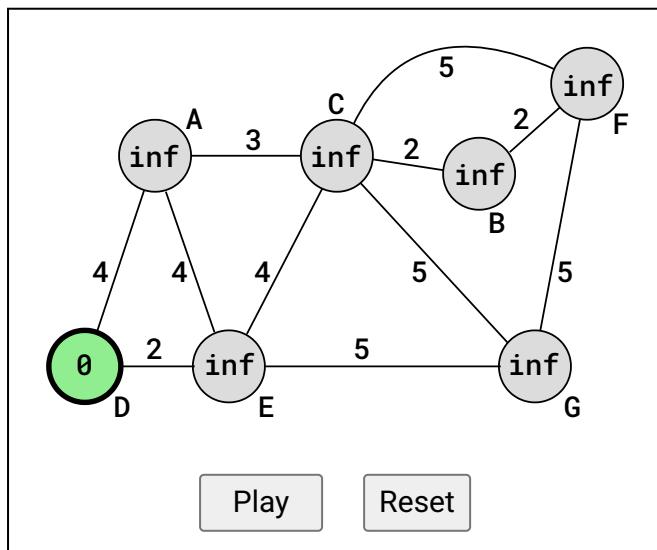
1. Set initial distances for all vertices: 0 for the source vertex, and infinity for all the other.
2. Choose the unvisited vertex with the shortest distance from the start to be the current vertex. So the algorithm will always start with the source as the current vertex.
3. For each of the current vertex's unvisited neighbor vertices, calculate the distance from the source and update the distance if the new, calculated, distance is lower.
4. We are now done with the current vertex, so we mark it as visited. A visited vertex is not checked again.

In the animation above, when a vertex is marked as visited, the vertex and its edges become faded to indicate that Dijkstra's algorithm is now done with that vertex, and will not visit it again.

**Note:** This basic version of Dijkstra's algorithm gives us the value of the shortest path cost to every vertex, but not what the actual path is. So for example, in the animation above, we get the shortest path cost value 10 to vertex F, but the algorithm does not give us which vertices (D->E->C->D->F) that make up this shortest path. We will add this functionality further down here on this page.

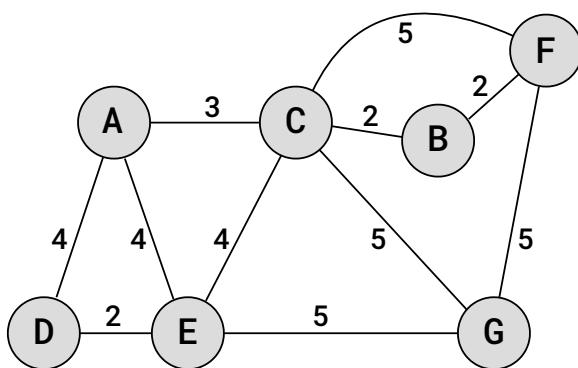
## A Detailed Dijkstra Simulation

Run the simulation below to get a more detailed understanding of how Dijkstra's algorithm runs on a specific graph, finding the shortest distances from vertex D.



This simulation shows how distances are calculated from vertex D to all other vertices, by always choosing the next vertex to be the closest unvisited vertex from the starting point.

Follow the step-by-step description below to get all the details of how Dijkstra's algorithm calculates the shortest distances.



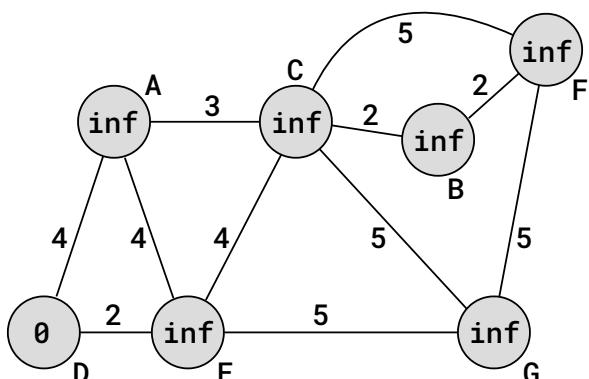
We want to find the shortest path from the source vertex D to all other vertices, so that for example the shortest path to C is D->E->C, with path weight  $2+4=6$ .

To find the shortest path, Dijkstra's algorithm uses an array with the distances to all other vertices, and initially sets these distances to infinite, or a very big number. And the distance to the vertex we start from (the source) is set to 0.

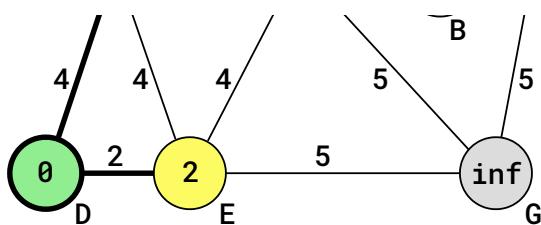
```

distances = [inf, inf, inf, 0, inf, inf, inf]
vertices   = [ A ,  B ,  C ,  D,  E ,  F ,  G ]
  
```

The image below shows the initial infinite distances to other vertices from the starting vertex D. The distance value for vertex D is 0 because that is the starting point.

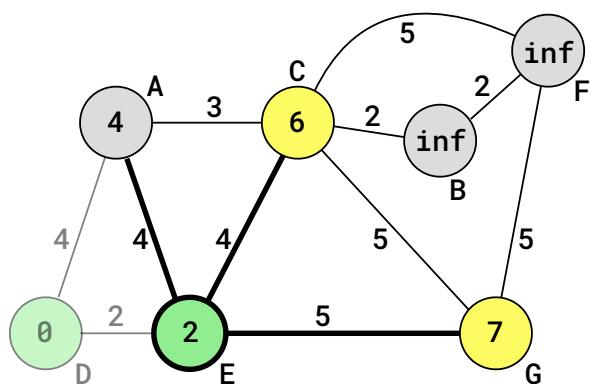


Dijkstra's algorithm then sets vertex D as the current vertex, and looks at the distance to the adjacent vertices. Since the initial distance to vertices A and E is infinite, the new distance to these are updated with the edge weights. So vertex A gets the distance changed from inf to 4, and vertex E gets the distance changed to 2. As mentioned on the previous page, updating the distance values in this way is called 'relaxing'.



After relaxing vertices A and E, vertex D is considered visited, and will not be visited again.

The next vertex to be chosen as the current vertex must be the vertex with the shortest distance to the source vertex (vertex D), among the previously unvisited vertices. Vertex E is therefore chosen as the current vertex after vertex D.



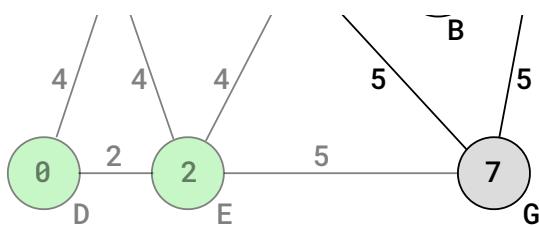
The distance to all adjacent and not previously visited vertices from vertex E must now be calculated, and updated if needed.

The calculated distance from D to vertex A, via E, is  $2+4=6$ . But the current distance to vertex A is already 4, which is lower, so the distance to vertex A is not updated.

The distance to vertex C is calculated to be  $2+4=6$ , which is less than infinity, so the distance to vertex C is updated.

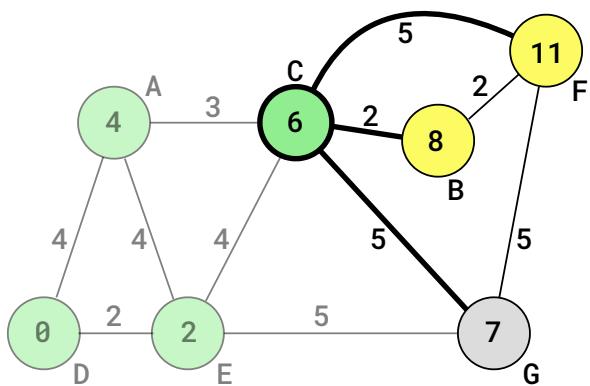
Similarly, the distance to node G is calculated and updated to be  $2+5=7$ .

The next vertex to be visited is vertex A because it has the shortest distance from D of all the unvisited vertices.



The calculated distance to vertex C, via A, is  $4+3=7$ , which is higher than the already set distance to vertex C, so the distance to vertex C is not updated.

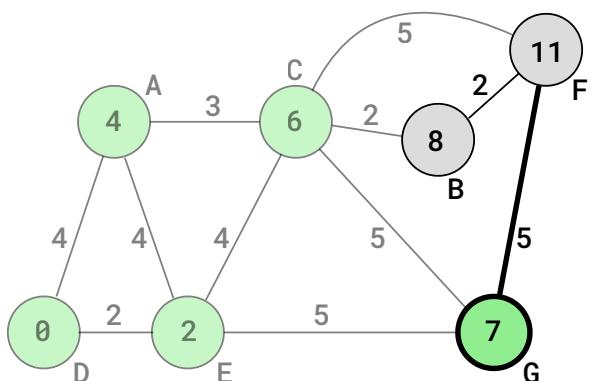
Vertex A is now marked as visited, and the next current vertex is vertex C because that has the lowest distance from vertex D between the remaining unvisited vertices.



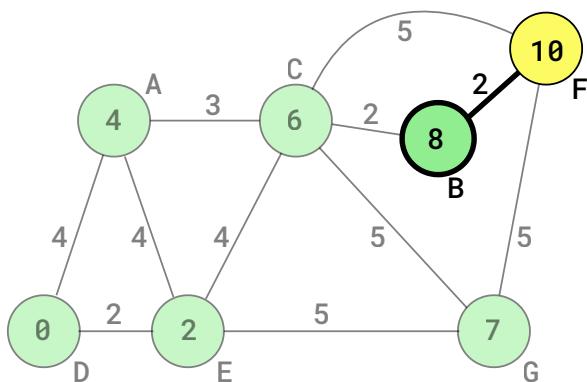
Vertex F gets updated distance  $6+5=11$ , and vertex B gets updated distance  $6+2=8$ .

Calculated distance to vertex G via vertex C is  $6+5=11$  which is higher than the already set distance of 7, so distance to vertex G is not updated.

Vertex C is marked as visited, and the next vertex to be visited is G because it has the lowest distance between the remaining unvisited vertices.



distance of the remaining unvisited vertices.



The new distance to F via B is  $8+2=10$ , because it is lower than F's existing distance of 11.

Vertex B is marked as visited, and there is nothing to check for the last unvisited vertex F, so Dijkstra's algorithm is finished.

Every vertex has been visited only once, and the result is the lowest distance from the source vertex D to every other vertex in the graph.

## Implementation of Dijkstra's Algorithm

To implement Dijkstra's algorithm, we create a `Graph` class. The `Graph` represents the graph with its vertices and edges:

```
class Graph:
    def __init__(self, size):
        self.adj_matrix = [[0] * size for _ in range(size)]
        self.size = size
        self.vertex_data = [''] * size

    def add_edge(self, u, v, weight):
        if 0 <= u < self.size and 0 <= v < self.size:
            self.adj_matrix[u][v] = weight
            self.adj_matrix[v][u] = weight # For undirected graph

    def add_vertex_data(self, vertex, data):
```

**Line 3:** We create the `adj_matrix` to hold all the edges and edge weights. Initial values are set to `0`.

**Line 4:** `size` is the number of vertices in the graph.

**Line 5:** The `vertex_data` holds the names of all the vertices.

**Line 7-10:** The `add_edge` method is used to add an edge from vertex `u` to vertex `v`, with edge weight `weight`.

**Line 12-14:** The `add_vertex_data` method is used to add a vertex to the graph. The index where the vertex should belong is given with the `vertex` argument, and `data` is the name of the vertex.

The `Graph` class also contains the method that runs Dijkstra's algorithm:

```
def dijkstra(self, start_vertex_data):
    start_vertex = self.vertex_data.index(start_vertex_data)
    distances = [float('inf')] * self.size
    distances[start_vertex] = 0
    visited = [False] * self.size

    for _ in range(self.size):
        min_distance = float('inf')
        u = None
        for i in range(self.size):
            if not visited[i] and distances[i] < min_distance:
                min_distance = distances[i]
                u = i

        if u is None:
            break

        visited[u] = True

        for v in range(self.size):
            if self.adj_matrix[u][v] != 0 and not visited[v]:
                alt = distances[u] + self.adj_matrix[u][v]
                if alt < distances[v]:
                    distances[v] = alt
```

**Line 18-19:** The initial distance is set to infinity for all vertices in the `distances` array, except for the start vertex, where the distance is 0.

**Line 20:** All vertices are initially set to `False` to mark them as not visited in the `visited` array.

**Line 23-28:** The next current vertex is found. Outgoing edges from this vertex will be checked to see if shorter distances can be found. It is the unvisited vertex with the lowest distance from the start.

**Line 30-31:** If the next current vertex has not been found, the algorithm is finished. This means that all vertices that are reachable from the source have been visited.

**Line 33:** The current vertex is set as visited before relaxing adjacent vertices. This is more effective because we avoid checking the distance to the current vertex itself.

**Line 35-39:** Distances are calculated for not visited adjacent vertices, and updated if the new calculated distance is lower.

After defining the `Graph` class, the vertices and edges must be defined to initialize the specific graph, and the complete code for this Dijkstra's algorithm example looks like this:

## Example

Python:

```
class Graph:
    def __init__(self, size):
        self.adj_matrix = [[0] * size for _ in range(size)]
        self.size = size
        self.vertex_data = [''] * size

    def add_edge(self, u, v, weight):
        if 0 <= u < self.size and 0 <= v < self.size:
            self.adj_matrix[u][v] = weight
            self.adj_matrix[v][u] = weight # For undirected graph

    def add_vertex_data(self, vertex, data):
        if 0 <= vertex < self.size:
            self.vertex_data[vertex] = data

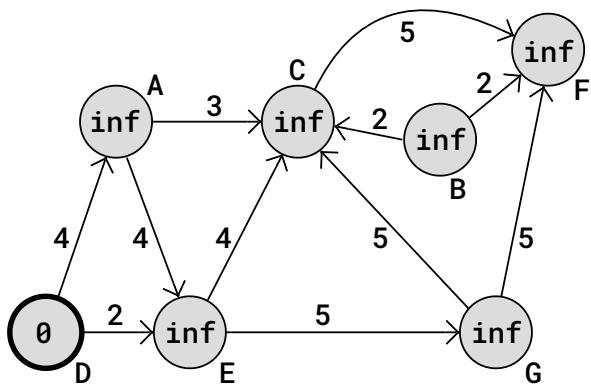
    def dijkstra(self, start_vertex_data):
```

# Dijkstra's Algorithm on Directed Graphs

To run Dijkstra's algorithm on directed graphs, very few changes are needed.

Similarly to the change we needed for [cycle detection for directed graphs](#), we just need to remove one line of code so that the adjacency matrix is not symmetric anymore.

Let's implement this directed graph and run Dijkstra's algorithm from vertex D.



Here is the implementation of Dijkstra's algorithm on the directed graph, with D as the source vertex:

## Example

Python:

```
class Graph:
    def __init__(self, size):
        self.adj_matrix = [[0] * size for _ in range(size)]
        self.size = size
        self.vertex_data = [''] * size

    def add_edge(self, u, v, weight):
        if 0 <= u < self.size and 0 <= v < self.size:
            self.adj_matrix[u][v] = weight
            #self.adj_matrix[v][u] = weight    For undirected graph

    def add_vertex_data(self, vertex, data):
```

```
        start_vertex = self.vertex_data.index(start_vertex_data)
        distances = [float('inf')] * self.size
        distances[start_vertex] = 0
        visited = [False] * self.size

        for _ in range(self.size):
            min_distance = float('inf')
            u = None
            for i in range(self.size):
                if not visited[i] and distances[i] < min_distance:
                    min_distance = distances[i]
                    u = i

            if u is None:
                break

            visited[u] = True

            for v in range(self.size):
                if self.adj_matrix[u][v] != 0 and not visited[v]:
                    alt = distances[u] + self.adj_matrix[u][v]
                    if alt < distances[v]:
                        distances[v] = alt

        return distances

g = Graph(7)

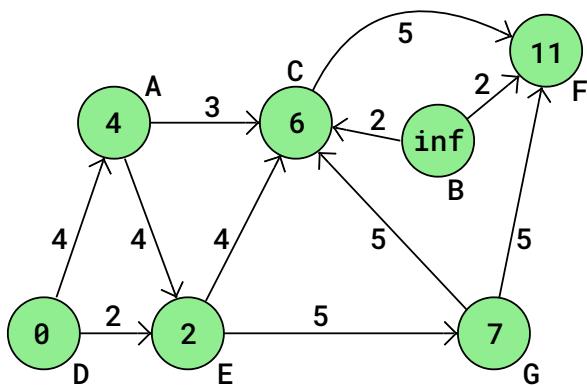
g.add_vertex_data(0, 'A')
g.add_vertex_data(1, 'B')
g.add_vertex_data(2, 'C')
g.add_vertex_data(3, 'D')
g.add_vertex_data(4, 'E')
g.add_vertex_data(5, 'F')
g.add_vertex_data(6, 'G')

g.add_edge(3, 0, 4) # D -> A, weight 5
g.add_edge(3, 4, 2) # D -> E, weight 2
g.add_edge(0, 2, 3) # A -> C, weight 3
g.add_edge(0, 4, 4) # A -> E, weight 4
g.add_edge(4, 2, 4) # E -> C, weight 4
g.add_edge(4, 6, 5) # E -> G, weight 5
```

```
64
65 # Dijkstra's algorithm from D to all vertices
66 print("Dijkstra's Algorithm starting from vertex D:\n")
67 distances = g.dijkstra('D')
68 for i, d in enumerate(distances):
    print(f"Shortest distance from D to {g.vertex_data[i]}: {d}")
```

[Run Example »](#)

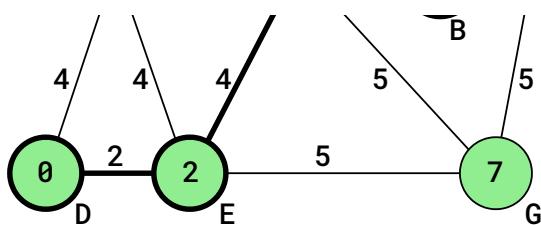
The image below shows us the shortest distances from vertex D as calculated by Dijkstra's algorithm.



This result is similar to the previous example using Dijkstra's algorithm on the undirected graph. However, there's a key difference: in this case, vertex B cannot be visited from D, and this means that the shortest distance from D to F is now 11, not 10, because the path can no longer go through vertex B.

## Returning The Paths from Dijkstra's Algorithm

With a few adjustments, the actual shortest paths can also be returned by Dijkstra's algorithm, in addition to the shortest path values. So for example, instead of just returning that the shortest path value is 10 from vertex D to F, the algorithm can also return that the shortest path is "D->E->C->B->F".



To return the path, we create a `predecessors` array to keep the previous vertex in the shortest path for each vertex. The `predecessors` array can be used to backtrack to find the shortest path for every vertex.

## Example

Python:

```
class Graph:
    # ... (rest of the Graph class)

    def dijkstra(self, start_vertex_data):
        start_vertex = self.vertex_data.index(start_vertex_data)
        distances = [float('inf')] * self.size
        predecessors = [None] * self.size
        distances[start_vertex] = 0
        visited = [False] * self.size

        for _ in range(self.size):
            min_distance = float('inf')
            u = None
            for i in range(self.size):
                if not visited[i] and distances[i] < min_distance:
                    min_distance = distances[i]
                    u = i

            if u is None:
                break

            visited[u] = True

            for v in range(self.size):
                if self.adj_matrix[u][v] != 0 and not visited[v]:
                    alt = distances[u] + self.adj_matrix[u][v]
```

```
31     return distances, predecessors
32
43
44 g = Graph(7)
45
46 # ... (rest of the graph setup)
47
48 # Dijkstra's algorithm from D to all vertices
49 print("Dijkstra's Algorithm starting from vertex D:\n")
50 distances, predecessors = g.dijkstra('D')
51 for i, d in enumerate(distances):
52     path = g.get_path(predecessors, 'D', g.vertex_data[i])
53     print(f"{path}, Distance: {d}")
```

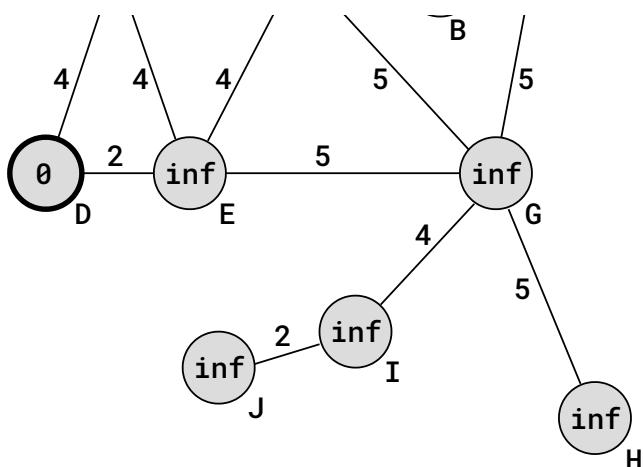
[Run Example »](#)

**Line 7 and 29:** The `predecessors` array is first initialized with `None` values, then it is updated with the correct predecessor for each vertex as the shortest path values are updated.

**Line 33-42:** The `get_path` method uses the `predecessors` array and returns a string with the shortest path from start to end vertex.

## Dijkstra's Algorithm with a Single Destination Vertex

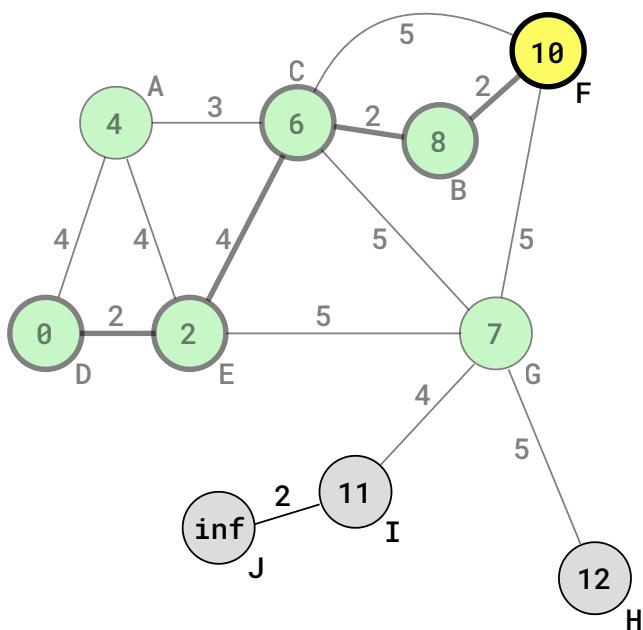
Let's say we are only interested in finding the shortest path between two vertices, like finding the shortest distance between vertex D and vertex F in the graph below.



Dijkstra's algorithm is normally used for finding the shortest path from one source vertex to all other vertices in the graph, but it can also be modified to only find the shortest path from the source to a single destination vertex, by just stopping the algorithm when the destination is reached (visited).

This means that for the specific graph in the image above, Dijkstra's algorithm will stop after visiting F (the destination vertex), before visiting vertices H, I and J because they are farther away from D than F is.

Below we can see the status of the calculated distances when Dijkstra's algorithm has found the shortest distance from D to F, and stops running.



In the image above, vertex F has just got updated with distance 10 from vertex B. Since F is the unvisited vertex with the lowest distance from D, it would normally be the next current vertex, but since it is the destination, the algorithm stops. If the algorithm did not stop, J would be the next vertex to get an updated distance  $11+2=13$ , from vertex I.

## Example

Python:

```
class Graph:
    # ... (existing methods)

    def dijkstra(self, start_vertex_data, end_vertex_data):
        start_vertex = self.vertex_data.index(start_vertex_data)
        end_vertex = self.vertex_data.index(end_vertex_data)
        distances = [float('inf')] * self.size
        predecessors = [None] * self.size
        distances[start_vertex] = 0
        visited = [False] * self.size

        for _ in range(self.size):
            min_distance = float('inf')
            u = None
            for i in range(self.size):
                if not visited[i] and distances[i] < min_distance:
                    min_distance = distances[i]
                    u = i

            if u is None or u == end_vertex:
                print(f"Breaking out of loop. Current vertex: {self.vertex_data[u]}")
                print(f"Distances: {distances}")
                break

            visited[u] = True
            print(f"Visited vertex: {self.vertex_data[u]}")

            for v in range(self.size):
                if self.adj_matrix[u][v] != 0 and not visited[v]:
                    alt = distances[u] + self.adj_matrix[u][v]
                    if alt < distances[v]:
                        distances[v] = alt
                        predecessors[v] = u

        return distances[end_vertex], self.get_path(predecessors,
# Example usage
g = Graph(7)
```

[Run Example »](#)

**Line 20-23:** If we are about to choose the destination vertex as the current vertex and mark it as visited, it means we have already calculated the shortest distance to the destination vertex, and Dijkstra's algorithm can be stopped in this single destination case.

## Time Complexity for Dijkstra's Algorithm

With  $V$  as the number of vertices in our graph, the time complexity for Dijkstra's algorithm is

$$O(V^2)$$

The reason why we get this time complexity is that the vertex with the lowest distance must be searched for to choose the next current vertex, and that takes  $O(V)$  time. And since this must be done for every vertex connected to the source, we need to factor that in, and so we get time complexity  $O(V^2)$  for Dijkstra's algorithm.

By using a Min-heap or Fibonacci-heap data structure for the distances instead (not yet explained in this tutorial), the time needed to search for the minimum distance vertex is reduced from  $O(V)$  to  $O(\log V)$ , which results in an improved time complexity for Dijkstra's algorithm

$$O(V \cdot \log V + E)$$

Where  $V$  is the number of vertices in the graph, and  $E$  is the number of edges.

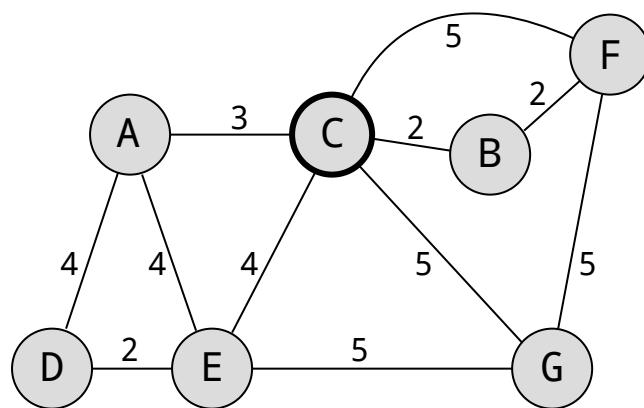
The improvement we get from using a Min-heap data structure for Dijkstra's algorithm is especially good if we have a large and sparse graph, which means a graph with a large number of vertices, but not as many edges.

The implementation of Dijkstra's algorithm with the Fibonacci-heap data structure is better for dense graphs, where each vertex has an edge to almost every other vertex.

## DSA Exercises

### Test Yourself With Exercises

Using Dijkstra's algorithm to find the shortest paths from vertex C in this graph:



What is the next vertex to be visited after C is visited?

Using Dijkstra's algorithm,  
the next vertex to be visited  
after vertex C is vertex .

[Submit Answer »](#)

[Start the Exercise](#)

 Previous

Next 

W3schools Pathfinder  
Track your progress - it's free!

[Sign Up](#) [Log in](#)

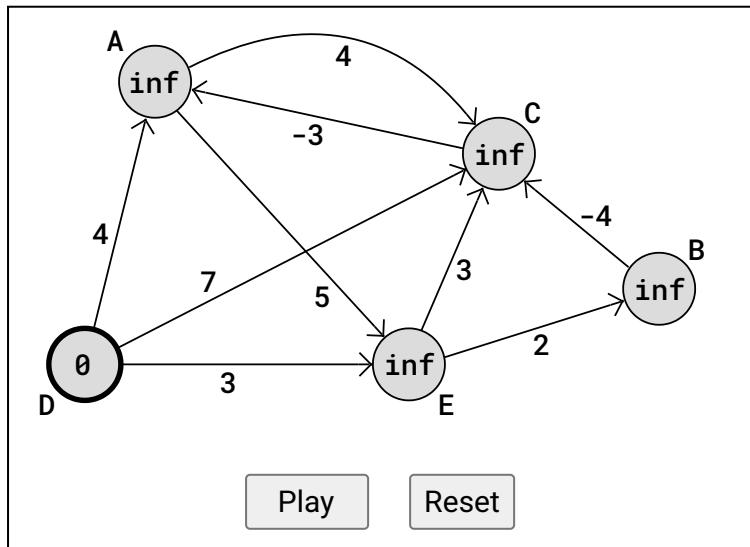
# DSA Bellman-Ford Algorithm

[« Previous](#)[Next »](#)

## The Bellman-Ford Algorithm

The Bellman-Ford algorithm is best suited to find the shortest paths in a directed graph, with one or more negative edge weights, from the source vertex to all other vertices.

It does so by repeatedly checking all the edges in the graph for shorter paths, as many times as there are vertices in the graph (minus 1).



The Bellman-Ford algorithm can also be used for graphs with positive edges (both directed and undirected), like we can with Dijkstra's algorithm, but Dijkstra's algorithm is preferred in such cases because it is faster.

Using the Bellman-Ford algorithm on a graph with negative cycles will not produce a result of shortest paths because in a negative cycle we can always go one more round and get a shorter path.

## How it works:

1. Set initial distance to zero for the source vertex, and set initial distances to infinity for all other vertices.
2. For each edge, check if a shorter distance can be calculated, and update the distance if the calculated distance is shorter.
3. Check all edges (step 2)  $V - 1$  times. This is as many times as there are vertices ( $V$ ), minus one.
4. Optional: Check for negative cycles. This will be explained in better detail later.

The animation of the Bellman-Ford algorithm above only shows us when checking of an edge leads to an updated distance, not all the other edge checks that do not lead to updated distances.

## Manual Run Through

The Bellman-Ford algorithm is actually quite straight forward, because it checks all edges, using the adjacency matrix. Each check is to see if a shorter distance can be made by going from the vertex on one side of the edge, via the edge, to the vertex on the other side of the edge.

And this check of all edges is done  $V - 1$  times, with  $V$  being the number of vertices in the graph.

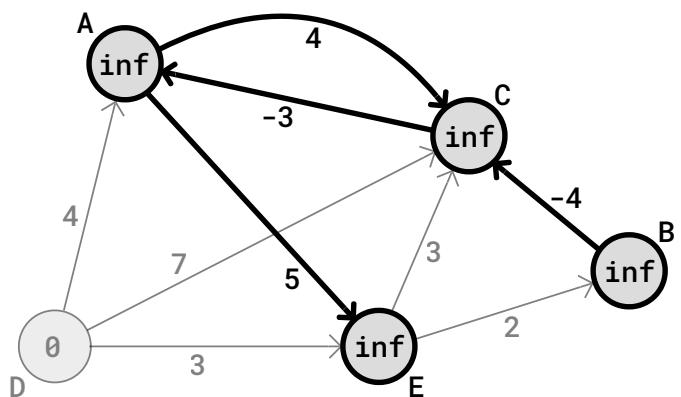
This is how the Bellman-Ford algorithm checks all the edges in the adjacency matrix in our graph  $5-1=4$  times:

TLIN SASS VUE DSA GEN AI SCIPY AWS CYBERSECURITY DATA SC

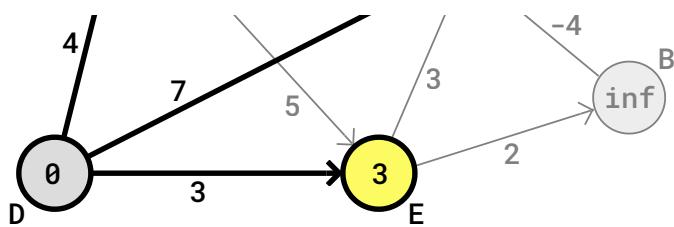
	A	B	C	D	E
A			4		5
B			-4		
C	-3				
D	4		7		3
E		2	3		

Checked all edges 0 times.

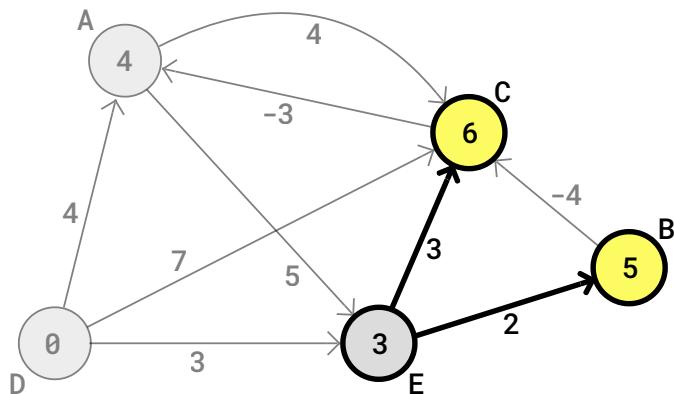
The first four edges that are checked in our graph are A->C, A->E, B->C, and C->A. These first four edge checks do not lead to any updates of the shortest distances because the starting vertex of all these edges has an infinite distance.



After the edges from vertices A, B, and C are checked, the edges from D are checked. Since the starting point (vertex D) has distance 0, the updated distances for A, B, and C are the edge weights going out from vertex D.

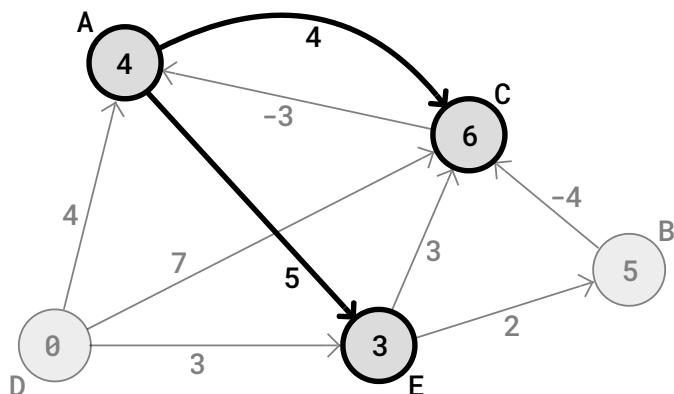


The next edges to be checked are the edges going out from vertex E, which leads to updated distances for vertices B and C.

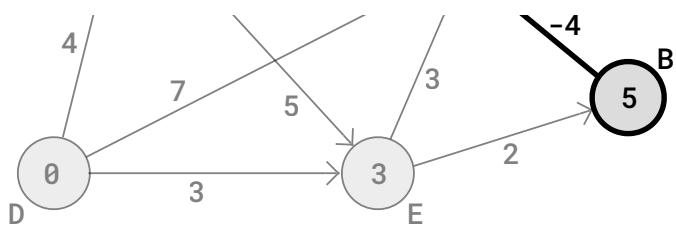


The Bellman-Ford algorithm have now checked all edges 1 time. The algorithm will check all edges 3 more times before it is finished, because Bellman-Ford will check all edges as many times as there are vertices in the graph, minus 1.

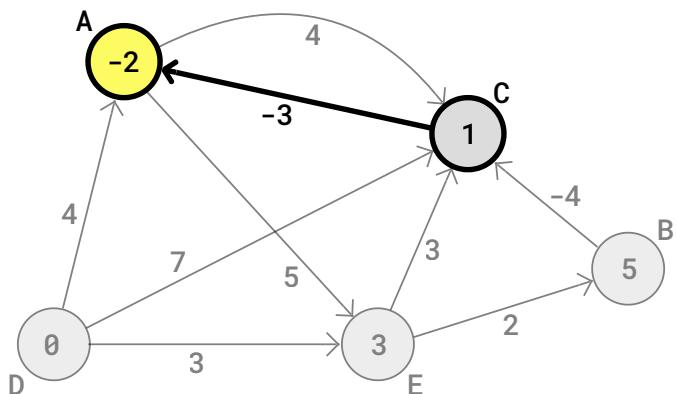
The algorithm starts checking all edges a second time, starting with checking the edges going out from vertex A. Checking the edges A->C and A->E do not lead to updated distances.



The next edge to be checked is B->C, going out from vertex B. This leads to an updated distance from vertex D to C of  $5-4=1$ .



Checking the next edge C->A, leads to an updated distance 1-3=-2 for vertex A.



The check of edge C->A in round 2 of the Bellman-Ford algorithm is actually the last check that leads to an updated distance for this specific graph. The algorithm will continue to check all edges 2 more times without updating any distances.

Checking all edges  $V - 1$  times in the Bellman-Ford algorithm may seem like a lot, but it is done this many times to make sure that the shortest distances will always be found.

## Implementation of The Bellman-Ford Algorithm

Implementing the Bellman-Ford algorithm is very similar to [how we implemented Dijkstra's algorithm](#).

We start by creating the `Graph` class, where the methods `__init__`, `add_edge`, and `add_vertex` will be used to create the specific graph we want to run the Bellman-Ford algorithm on to find the shortest paths.

TLIN    SASS    VUE    DSA    GEN AI    SCIPY    AWS    CYBERSECURITY    DATA SC

```
3     self.adj_matrix = [[0] * size for _ in range(size)]
4     self.size = size
5     self.vertex_data = [''] * size
6
7     def add_edge(self, u, v, weight):
8         if 0 <= u < self.size and 0 <= v < self.size:
9             self.adj_matrix[u][v] = weight
10            #self.adj_matrix[v][u] = weight # For undirected graph
11
12    def add_vertex_data(self, vertex, data):
13        if 0 <= vertex < self.size:
14            self.vertex_data[vertex] = data
```

The `bellman_ford` method is also placed inside the `Graph` class. It is this method that runs the Bellman-Ford algorithm.

```
16    def bellman_ford(self, start_vertex_data):
17        start_vertex = self.vertex_data.index(start_vertex_data)
18
19        for _ in range(self.size - 1):
20            for u in range(self.size):
21                for v in range(self.size):
22                    if self.adj_matrix[u][v] != 0:
23                        new_dist = self.vertex_data[u] + self.adj_matrix[u][v]
24                        if new_dist < self.vertex_data[v]:
25                            self.vertex_data[v] = new_dist
26
27        print(f'Relaxing edge {self.vertex_data[u]}-{self.vertex_data[v]}')
28
29        return distances
```

**Line 18-19:** At the beginning, all vertices are set to have an infinite long distance from the starting vertex, except for the starting vertex itself, where the distance is set to 0.

**Line 21:** All edges are checked  $V - 1$  times.

update the distance to that vertex  $v$ .

The complete code, including the initialization of our specific graph and code for running the Bellman-Ford algorithm, looks like this:

## Example

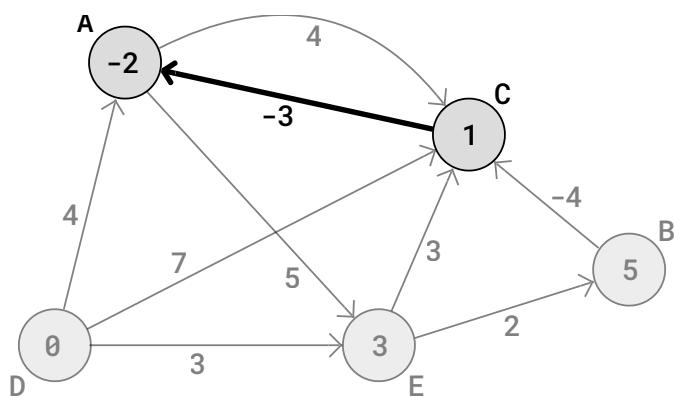
Python:

```
1 class Graph:
2     def __init__(self, size):
3         self.adj_matrix = [[0] * size for _ in range(size)]
4         self.size = size
5         self.vertex_data = [''] * size
6
7     def add_edge(self, u, v, weight):
8         if 0 <= u < self.size and 0 <= v < self.size:
9             self.adj_matrix[u][v] = weight
10            #self.adj_matrix[v][u] = weight # For undirected graph
11
12    def add_vertex_data(self, vertex, data):
13        if 0 <= vertex < self.size:
14            self.vertex_data[vertex] = data
15
16    def bellman_ford(self, start_vertex_data):
17        start_vertex = self.vertex_data.index(start_vertex_data)
```

[Run Example »](#)

## Negative Edges in The Bellman-Ford Algorithm

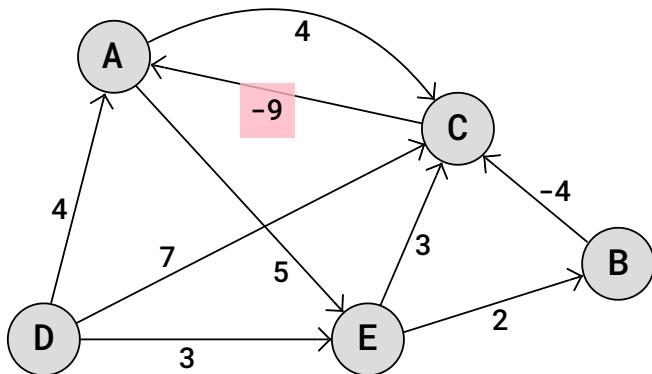
To say that the Bellman-Ford algorithm finds the "shortest paths" is not intuitive, because how can we draw or imagine distances that are negative? So, to make it easier to understand we could instead say that it is the "*cheapest paths*" that are found with Bellman-Ford.



With this interpretation in mind, the -3 weight on edge C->A could mean that the fuel cost is \$5 driving from C to A, and that we get paid \$8 for picking up packages in C and delivering them in A. So we end up earning \$3 more than we spend. Therefore, a total of \$2 can be made by driving the delivery route D->E->B->C->A in our graph above.

## Negative Cycles in The Bellman-Ford Algorithm

If we can go in circles in a graph, and the sum of edges in that circle is negative, we have a negative cycle.



By changing the weight on edge C->A from -3 to -9, we get two negative cycles: A->C->A and A->E->C->A. And every time we check these edges with the Bellman-Ford algorithm, the distances we calculate and update just become lower and lower.

The problem with negative cycles is that a shortest path does not exist, because we can always go one more round to get a path that is shorter.

That is why it is useful to implement the Bellman-Ford algorithm with detection for negative cycles.

# Ford Algorithm

After running the Bellman-Ford algorithm, checking all edges in a graph  $V - 1$  times, all the shortest distances are found.

But, if the graph contains negative cycles, and we go one more round checking all edges, we will find at least one shorter distance in this last round, right?

So to detect negative cycles in the Bellman-Ford algorithm, after checking all edges  $V - 1$  times, we just need to check all edges one more time, and if we find a shorter distance this last time, we can conclude that a negative cycle must exist.

Below is the `bellman_ford` method, with negative cycle detection included, running on the graph above with negative cycles due to the C->A edge weight of -9:

## Example

Python:

```
16     def bellman_ford(self, start_vertex_data):
17         start_vertex = self.vertex_data.index(start_vertex_data)
18         distances = [float('inf')] * self.size
19         distances[start_vertex] = 0
20
21         for i in range(self.size - 1):
22             for u in range(self.size):
23                 for v in range(self.size):
24                     if self.adj_matrix[u][v] != 0:
25                         if distances[u] + self.adj_matrix[u][v] <
26                             distances[v] = distances[u] + self.adj
27                         print(f"Relaxing edge {self.vertex_dat
28
```

**Line 30-33:** All edges are checked one more time to see if there are negative cycles.

**Line 34:** Returning `True` indicates that a negative cycle exists, and `None` is returned instead of the shortest distances, because finding the shortest distances in a graph with negative cycles does not make sense (because a shorter distance can always be found by checking all edges one more time).

**Line 36:** Returning `False` means that there is no negative cycles, and the `distances` can be returned.

## Returning The Paths from The Bellman-Ford Algorithm

We are currently finding the total weight of the the shortest paths, so that for example "Distance from D to A: -2" is a result from running the Bellman-Ford algorithm.

But by recording the predecessor of each vertex whenever an edge is relaxed, we can use that later in our code to print the result including the actual shortest paths. This means we can give more information in our result, with the actual path in addition to the path weight: "D->E->B->C->A, Distance: -2".

This last code example is the complete code for the Bellman-Ford algorithm, with everything we have discussed up until now: finding the weights of shortest paths, detecting negative cycles, and finding the actual shortest paths:

## Example

Python:

```
class Graph:
    def __init__(self, size):
        self.adj_matrix = [[0] * size for _ in range(size)]
        self.size = size
        self.vertex_data = [''] * size

    def add_edge(self, u, v, weight):
        if 0 <= u < self.size and 0 <= v < self.size:
            self.adj_matrix[u][v] = weight
```

TLIN   SASS   VUE   DSA   GEN AI   SCIPY   AWS   CYBERSECURITY   DATA SC

```
15     self.vertex_data[vertex] = data
16
17 def bellman_ford(self, start_vertex_data):
```

[Run Example »](#)

**Line 19:** The `predecessors` array holds each vertex' predecessor vertex in the shortest path.

**Line 28:** The `predecessors` array gets updated with the new predecessor vertex every time an edge is relaxed.

**Line 40-49:** The `get_path` method uses the `predecessors` array to generate the shortest path string for each vertex.

## Time Complexity for The Bellman-Ford Algorithm

The time complexity for the Bellman-Ford algorithm mostly depends on the nested loops.

**The outer for-loop** runs  $V - 1$  times, or  $V$  times in case we also have negative cycle detection. For graphs with many vertices, checking all edges one less time than there are vertices makes little difference, so we can say that the outer loop contributes with  $O(V)$  to the time complexity.

**The two inner for-loops** checks all edges in the graph. If we assume a worst case scenario in terms of time complexity, then we have a very dense graph where every vertex has an edge to every other vertex, so for all vertex  $V$  the edge to all other vertices  $V$  must be checked, which contributes with  $O(V^2)$  to the time complexity.

So in total, we get the time complexity for the Bellman-Ford algorithm:

$$O(V^3)$$

However, in practical situations and especially for sparse graphs, meaning each vertex only has edges to a small portion of the other vertices, time complexity of the two inner for-loops checking all edges can be approximated from  $O(V^2)$  to  $O(E)$ , and we get the total time complexity for Bellman-Ford:

$$O(V \cdot E)$$

61

## DSA Exercises

63

64

65

66

### Test Yourself With Exercises

67

68

69

70

#### Exercise:

72

In the adjacency matrix below:

73

74

	A	B	C	D	E
A			4		5
B			-4		
C	-3				
D	4		7		3
E		2	3		

What is the edge weight of the edge going from D to E?

The D->E edge weight is .

[Submit Answer »](#)

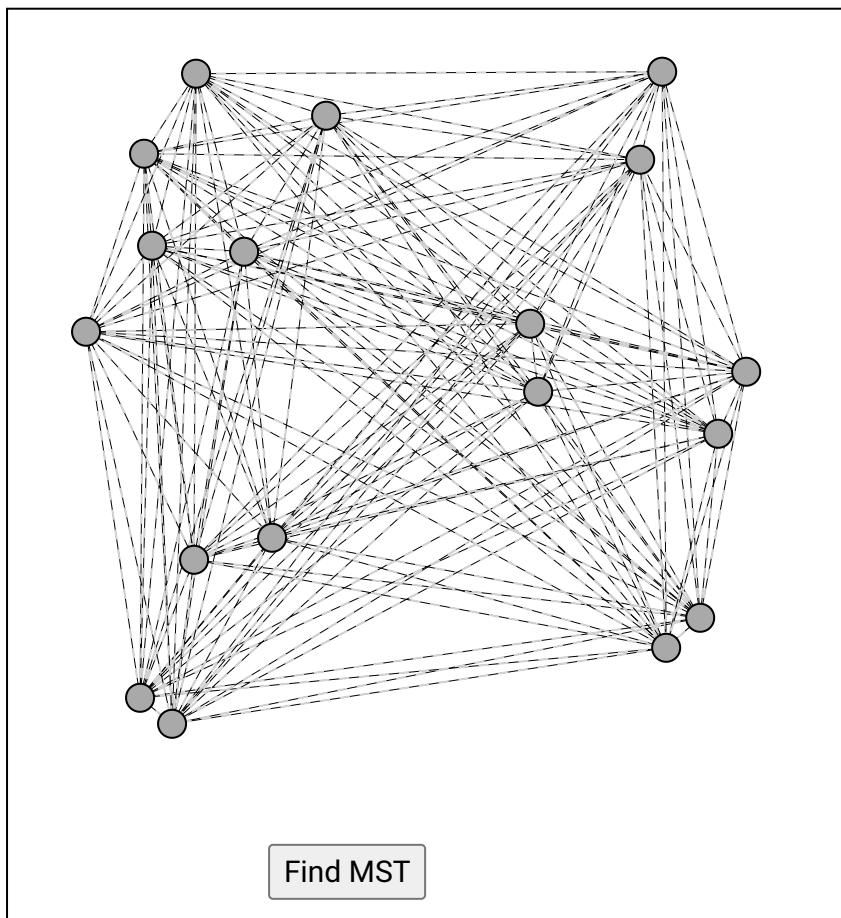
[Start the Exercise](#)

# DSA Minimum Spanning Tree

[Previous](#)[Next](#)

## The Minimum Spanning Tree Problem

The Minimum Spanning Tree (MST) is the collection of edges required to connect all vertices in an undirected graph, with the minimum total edge weight.



The animation above runs [Prim's algorithm](#) to find the MST. Another way to find the MST, which also works for unconnected graphs, is to run [Kruskal's algorithm](#).

connect houses to the internet or to the electrical grid, or it can help us finding the fastest route to deliver packages.

## An MST Thought Experiment

Let's imagine that the circles in the animation above are villages that are without electrical power, and you want to connect them to the electrical grid. After one village is given electrical power, the electrical cables must be spread out from that village to the others. The villages can be connected in a lot of different ways, each route having a different cost.

The electrical cables are expensive, and digging ditches for the cables, or stretching the cables in the air is expensive as well. The terrain can certainly be a challenge, and then there is perhaps a future cost for maintenance that is different depending on where the cables end up.

All these route costs can be factored in as edge weights in a graph. Every vertex represents a village, and every edge represents a possible route for the electrical cable between two villages.

After such a graph is created, the Minimum Spanning Tree (MST) can be found, and that will be the most effective way to connect these villages to the electrical grid.

And this is actually what the first MST algorithm (Borůvka's algorithm) was made for in 1926: To find the best way to connect the historical region of Moravia, in the Check Republic, to the electrical grid.

## MST Algorithms

The next two pages in this tutorial explains two algorithms that finds the Minimum Spanning Tree in a graph: [Prim's algorithm](#), and [Kruskal's algorithm](#).

	Prim's algorithm	Kruskal's algorithm
<i>Can it find MSTs (a Minimum Spanning Forest) in an unconnected graph?</i>	No	Yes
<i>How does it start?</i>	The MST grows from a randomly chosen vertex.	The first edge in the MST is the edge with lowest edge weight.

# DSA Prim's Algorithm

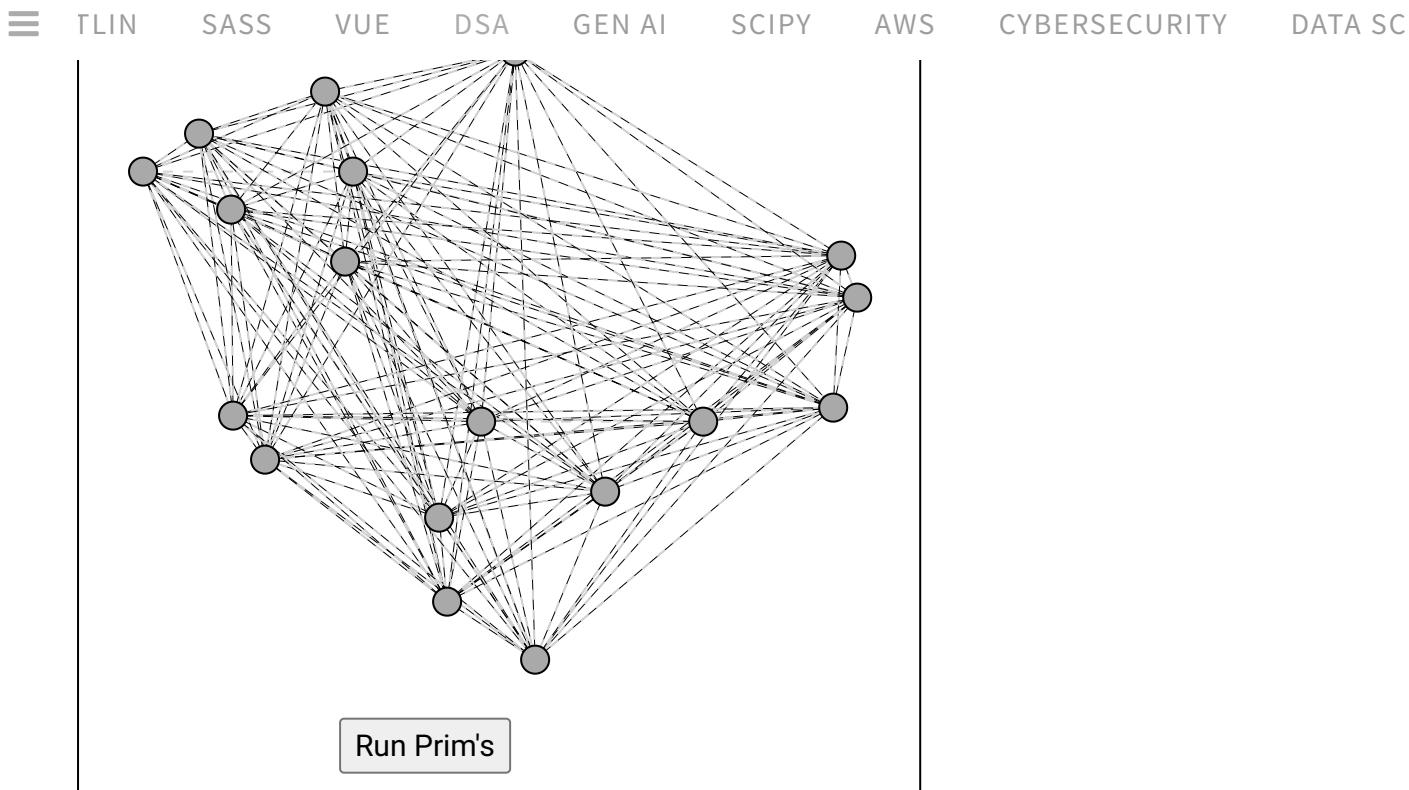
[« Previous](#)[Next »](#)

Prim's algorithm was invented in 1930 by the Czech mathematician Vojtěch Jarník.

The algorithm was then rediscovered by Robert C. Prim in 1957, and also rediscovered by Edsger W. Dijkstra in 1959. Therefore, the algorithm is also sometimes called "Jarník's algorithm", or the "Prim-Jarník algorithm".

## Prim's Algorithm

Prim's algorithm finds the Minimum Spanning Tree (MST) in a connected and undirected graph.



The MST found by Prim's algorithm is the collection of edges in a graph, that connects all vertices, with a minimum sum of edge weights.

Prim's algorithm finds the MST by first including a random vertex to the MST. The algorithm then finds the vertex with the lowest edge weight from the current MST, and includes that to the MST. Prim's algorithm keeps doing this until all nodes are included in the MST.

Prim's algorithm is greedy, and has a straightforward way to create a minimum spanning tree.

For Prim's algorithm to work, all the nodes must be connected. To find the MST's in an unconnected graph, [Kruskal's algorithm](#) can be used instead. You can read about Kruskal's algorithm on the next page.

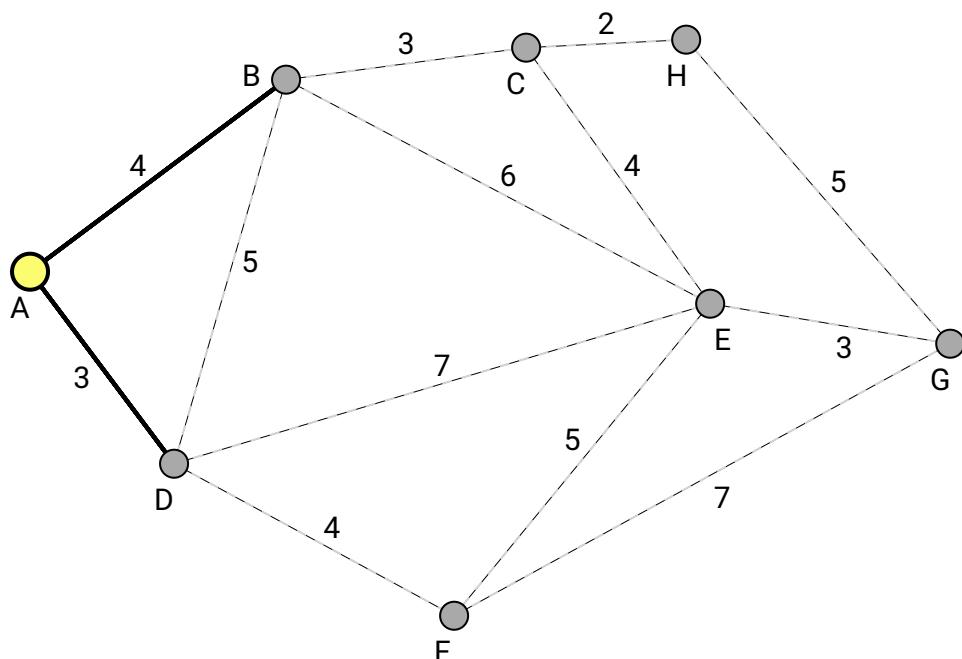
### How it works:

1. Choose a random vertex as the starting point, and include it as the first vertex in the MST.
2. Compare the edges going out from the MST. Choose the edge with the lowest weight that connects a vertex among the MST vertices to a vertex outside the MST.
3. Add that edge and vertex to the MST.
4. Keep doing step 2 and 3 until all vertices belong to the MST.

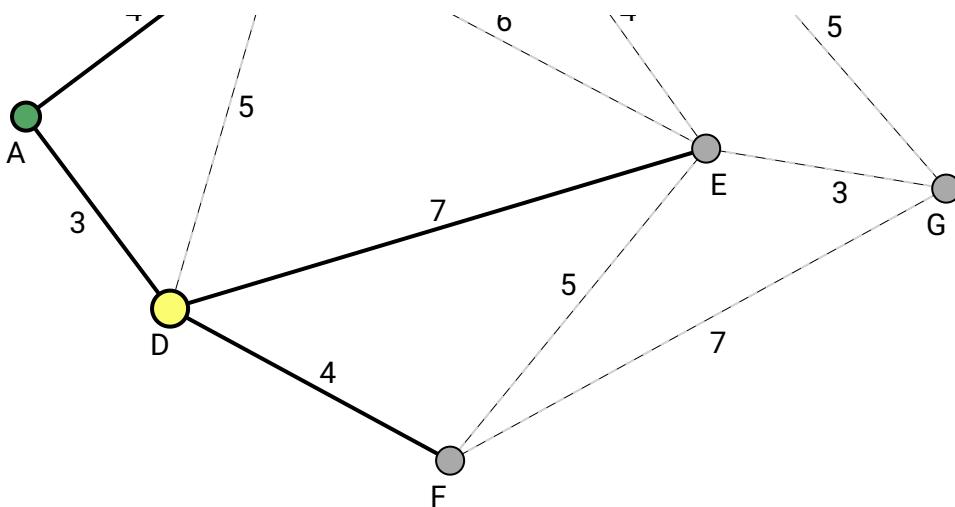
# Manual Run Through

Let's run through Prim's algorithm manually on the graph below, so that we understand the detailed step-by-step operations before we try to program it.

Prim's algorithm starts growing the Minimum Spanning Tree (MST) from a random vertex, but for this demonstration vertex A is chosen as the starting vertex.



From vertex A, the MST grows along the edge with the lowest weight. So vertices A and D now belong to the group of vertices that belong to the Minimum Spanning Tree.



A `parents` array is central to how Prim's algorithm grows the edges in the MST.

At this point, the `parents` array looks like this:

```
parents = [-1,  0, -1,  0,  3,  3, -1, -1]
#vertices [ A,  B,  C,  D,  E,  F,  G,  H]
```

Vertex A, the starting vertex, has no parent, and has therefore value `-1`. Vertex D's parent is A, that is why D's parent value is `0` (vertex A is located at index 0). B's parent is also A, and D is the parent of E and F.

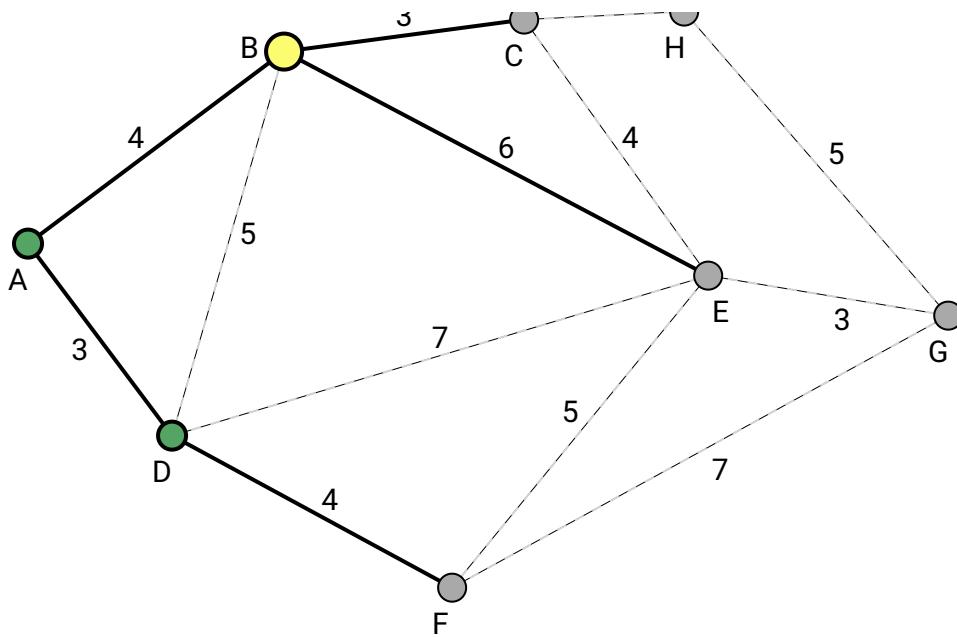
The `parents` array helps us to keep the MST tree structure (a vertex can only have one parent).

Also, to avoid cycles and to keep track of which vertices are currently in the MST, the `in_mst` array is used.

The `in_mst` array currently looks like this:

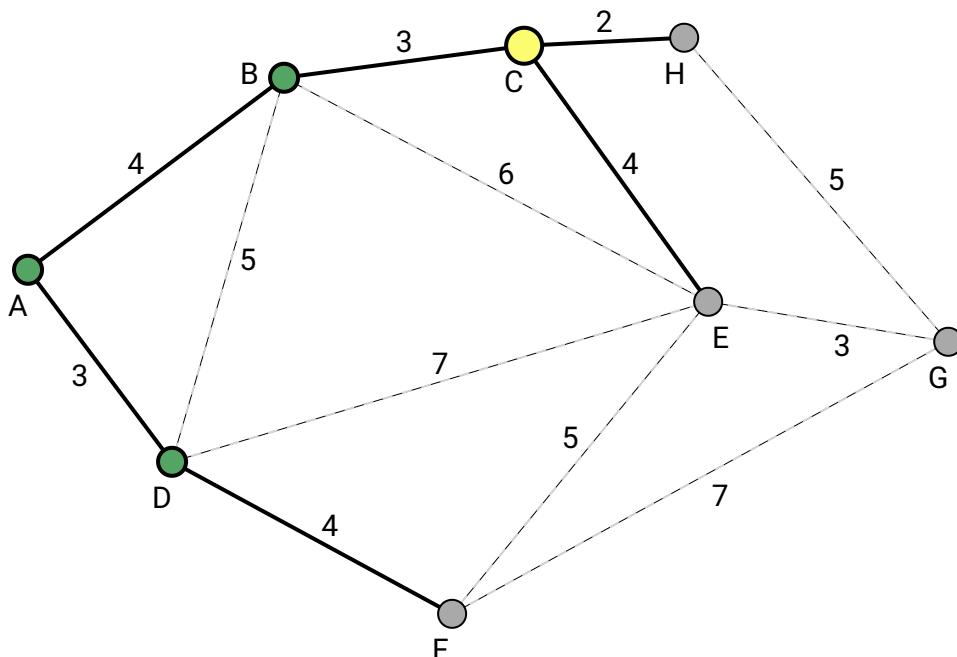
```
in_mst = [ true, false, false,  true, false, false, false, false]
#vertices [     A,      B,      C,      D,      E,      F,      G,      H]
```

The next step in Prim's algorithm is to include one more vertex as part of the MST, and the vertex closest to the current MST nodes A and D is chosen.



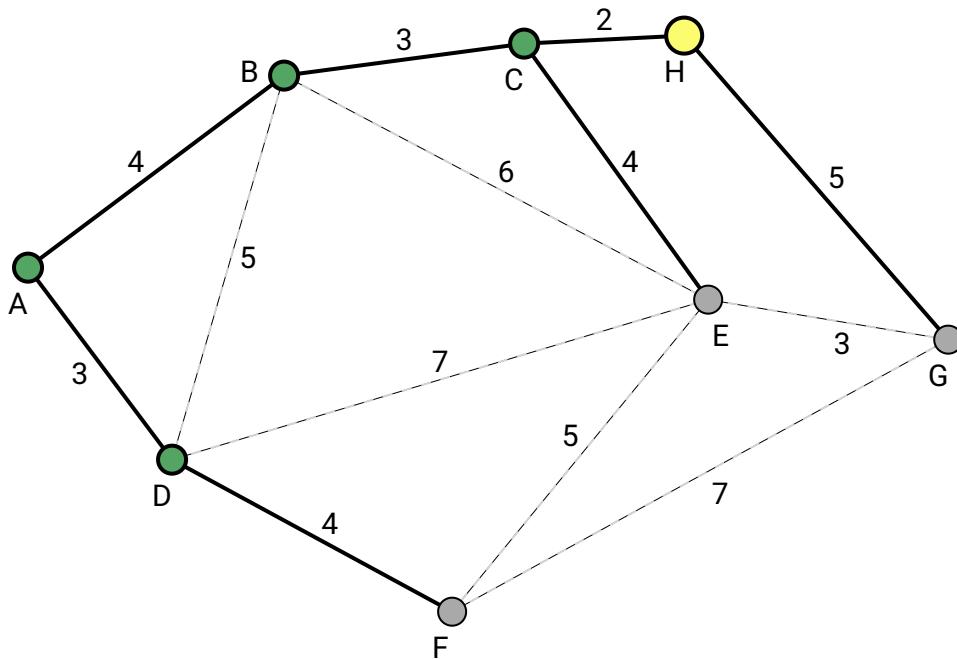
As you can see, the MST edge to E came from vertex D before, now it comes from vertex B, because B-E with weight **6** is lower than D-E with weight **7**. Vertex E can only have one parent in the MST tree structure (and in the `parents` array), so B-E and D-E cannot both be MST edges to E.

The next vertex in the MST is vertex C, because edge B-C with weight **3** is the shortest edge weight from the current MST vertices.



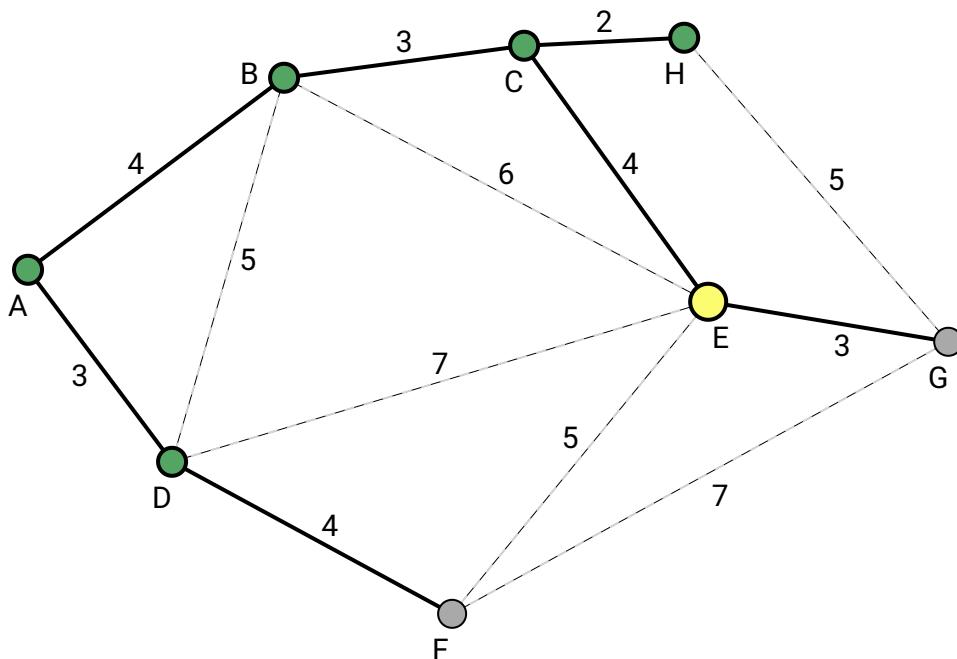
As vertex C is included in the MST, edges out from C are checked to see if there are edges with a lower weight from this MST vertex to vertices outside the MST. Edge C-E has a lower weight (**3**)

becomes the parent of vertex G in the `parents` array.

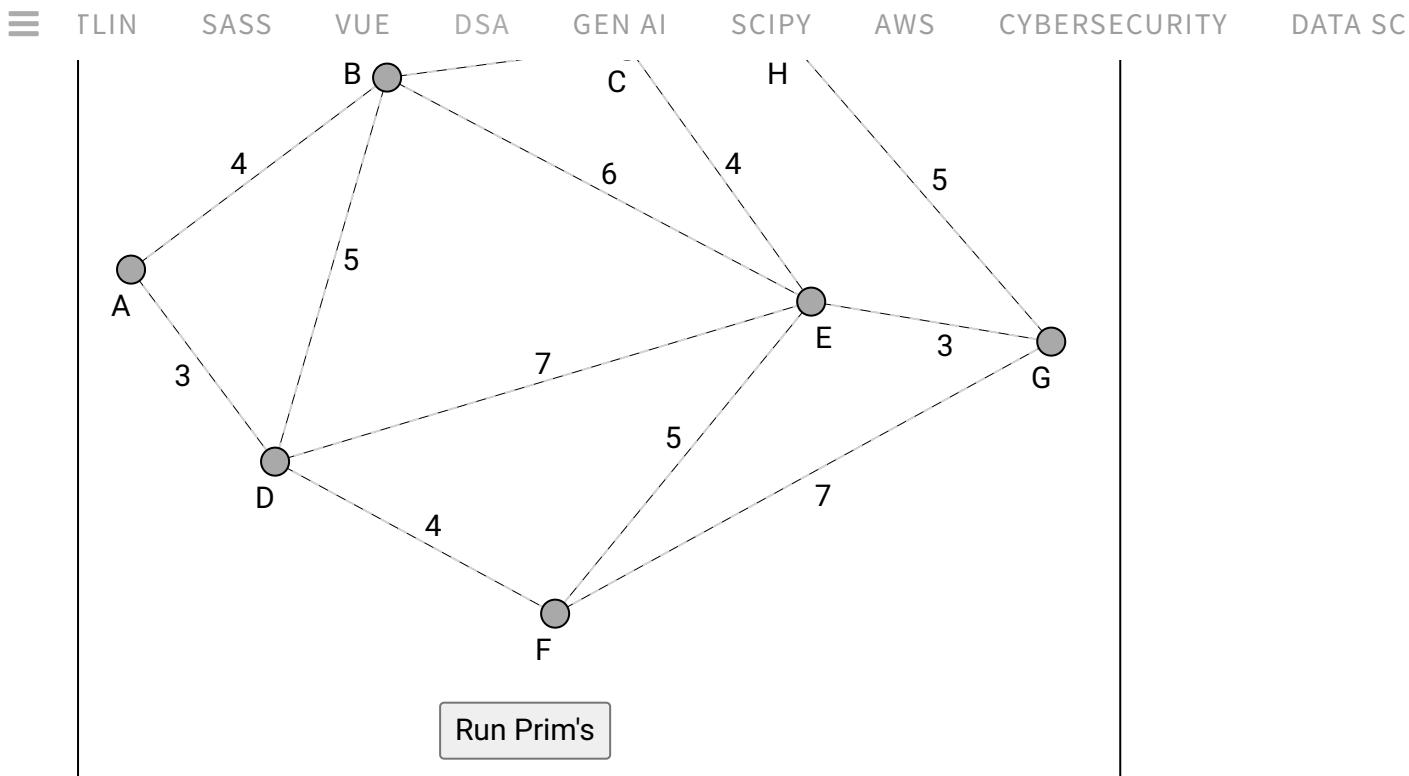


The next vertex to be included in the MST is either E or F because they have both the lowest edge weight to them: 4 .

We choose vertex E as the next vertex to be included in the MST for this demonstration.



The next and last two vertices to be added to the MST are vertices F and G. D-F is the MST edge to F, and E-G is the MST edge to G because these edges are the edges with the lowest weight from the current MST.



## Implementation of Prim's Algorithm

For Prim's algorithm to find a Minimum Spanning Tree (MST), we create a `Graph` class. We will use the methods inside this `Graph` class later to create the graph from the example above, and to run Prim's algorithm on it.

```
1 | class Graph:  
2 |     def __init__(self, size):
```

```
6 |
```

```
11 |
```

undirected graph.

**Line 12-14:** The `add_vertex_data` method is used for giving names to the vertices, like for example 'A' or 'B'.

Now that the structure for creating a graph is in place, we can implement Prim's algorithm as a method inside the `Graph` class:

```
16 |     def prims_algorithm(self):  
  
20 |  
  
22 |         print("Edge \tWeight")  
23 |         for _ in range(self.size):  
  
26 |             in_mst[u] = True  
27 |  
28 |             if parents[u] != -1: # Skip printing for the first ve  
29 |                 print(f"\t{self.vertex_data[parents[u]]} - {self.verte  
30 |  
31 |
```

**Line 17:** The `in_mst` array holds the status of which vertices are currently in the MST. Initially, none of the vertices are part of the MST.

**Line 18:** The `key_values` array holds the current shortest distance from the MST vertices to each vertex outside the MST.

**Line 19:** The MST edges are stored in the `parents` array. Each MST edge is stored by storing the parent index for each vertex.

**Line 25:** The index is found for the vertex with the lowest key value that is not yet part of the MST. Check out these explanations for [min](#) and [lambda](#) to better understand this Python code line.

**Line 32-35:** After a new vertex is added to the MST (line 27), this part of the code checks to see if there are now edges from this newly added MST vertex that can lower the key values to other vertices outside the MST. If that is the case, the `key_values` and `parents` arrays are updated accordingly. This can be seen clearly in the animation when a new vertex is added to the MST and becomes the active (current) vertex.

Now let's create the graph from the "Manual Run Through" above and run Prim's algorithm on it:

## Example

Python:

```
1  class Graph:
2      def __init__(self, size):
3          self.adj_matrix = [[0] * size for _ in range(size)]
4          self.size = size
5          self.vertex_data = [''] * size
6
7      def add_edge(self, u, v, weight):
8          if 0 <= u < self.size and 0 <= v < self.size:
9              self.adj_matrix[u][v] = weight
10             self.adj_matrix[v][u] = weight # For undirected graph
11
12     def add_vertex_data(self, vertex, data):
13         if 0 <= vertex < self.size:
14             self.vertex_data[vertex] = data
15
16     def prims_algorithm(self):
17         in_mst = [False] * self.size
```

[Run Example »](#)

**Line 32:** We can actually avoid the last loop in Prim's algorithm by changing this line to `for _ in range(self.size - 1):`. This is because when there is just one vertex not yet in the

# Time Complexity for Prim's Algorithm

For a general explanation of what time complexity is, visit [this page](#).

With  $V$  as the number of vertices in our graph, the time complexity for Prim's algorithm is

$$O(V^2)$$

The reason why we get this time complexity is because of the nested loops inside the Prim's algorithm (one for-loop with two other for-loops inside it).

The first for-loop (line 24) goes through all the vertices in the graph. This has time complexity  $O(V)$ .

The second for-loop (line 25) goes through all the adjacent vertices in the graph to find the vertex with the lowest key value that is outside the MST, so that it can be the next vertex included in the MST. This has time complexity  $O(V)$ .

After a new vertex is included in the MST, a third for-loop (line 32) checks all other vertices to see if there are outgoing edges from the newly added MST vertex to vertices outside the MST that can lead to lower key values and updated parent relations. This also has time complexity  $O(V)$ .

Putting the time complexities together we get:

$$\begin{aligned} O(V) \cdot (O(V) + O(V)) &= O(V) \cdot (2 \cdot O(V)) \\ &= O(V \cdot 2 \cdot V) \\ &= O(2 \cdot V^2) \\ &= O(V^2) \end{aligned}$$

By using a priority queue data structure to manage key values, instead of using an array like we do here, the time complexity for Prim's algorithm can be reduced to:

$$O(E \cdot \log V)$$

Where  $E$  is the number of edges in the graph, and  $V$  is the number of vertices.

Such an implementation of Prim's algorithm using a priority queue is best for sparse graphs. A graph is sparse when each vertex is just connected to a few of the other vertices.

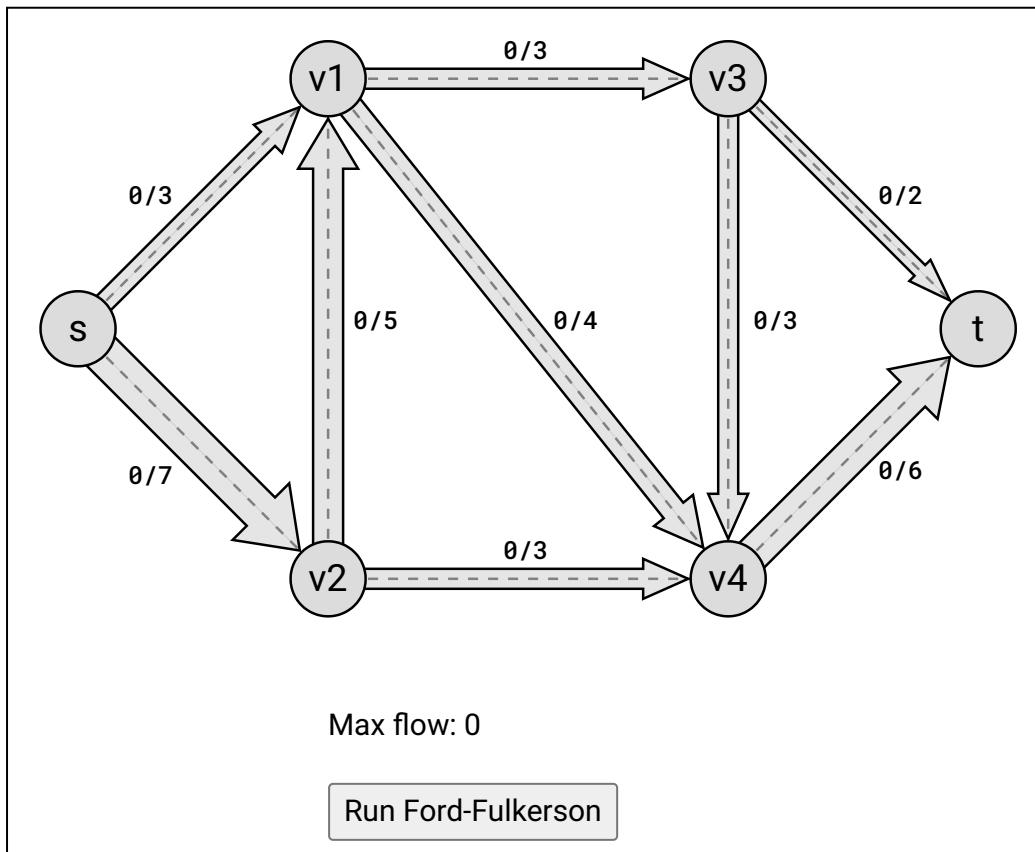
# DSA Maximum Flow

[« Previous](#)[Next »](#)

## The Maximum Flow Problem

The Maximum Flow problem is about finding the maximum flow through a directed graph, from one place in the graph to another.

More specifically, the flow comes from a source vertex  $s$ , and ends up in a sink vertex  $t$ , and each edge in the graph is defined with a flow and a capacity, where the capacity is the maximum flow that edge can have.



- To find out where in the flow network expanding the capacity will lead to the highest maximum flow, with the purpose of increasing for example traffic, data traffic, or water flow.

# Terminology And Concepts

A **flow network** is often what we call a directed graph with a flow flowing through it.

The **capacity**  $c$  of an edge tells us how much flow is allowed to flow through that edge.

Each edge also has a **flow** value that tells how much the current flow is in that edge.



The edge in the image above  $v_1 \rightarrow v_2$ , going from vertex  $v_1$  to vertex  $v_2$ , has its flow and capacity described as  $0/7$ , which means the flow is  $0$ , and the capacity is  $7$ . So the flow in this edge can be increased up to  $7$ , but not more.

In its simplest form, flow network has one **source vertex**  $s$  where the flow comes out, and one **sink vertex**  $t$  where the flow goes in. The other vertices just have flow passing through them.

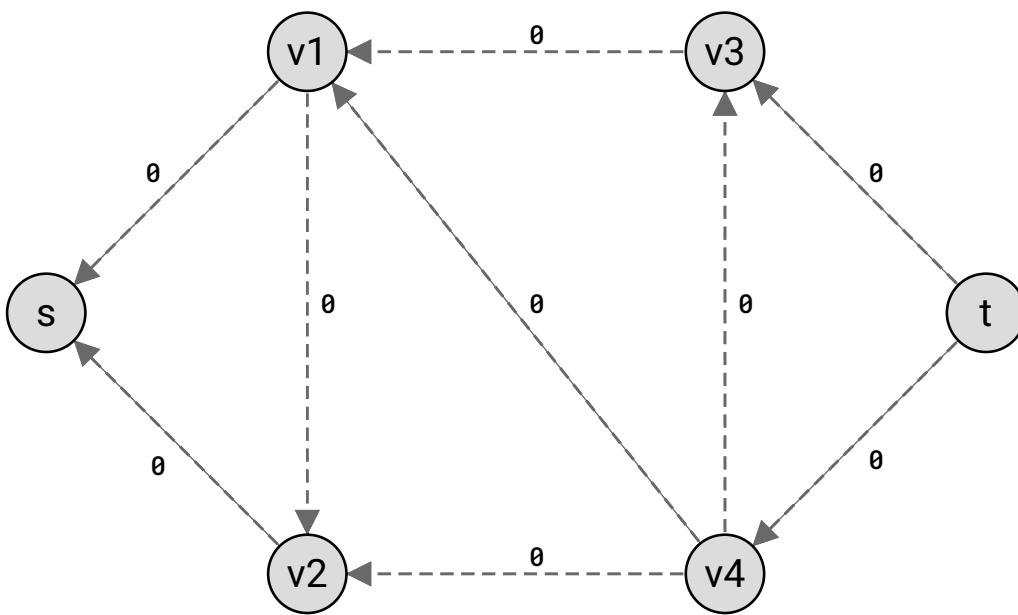
For all vertices except  $s$  and  $t$ , there is a **conservation of flow**, which means that the same amount of flow that goes into a vertex, must also come out of it.

The maximum flow is found by algorithms such as Ford-Fulkerson, or Edmonds-Karp, by sending more and more flow through the edges in the flow network until the capacity of the edges are such that no more flow can be sent through. Such a path where more flow can be sent through is called an **augmented path**.

The Ford-Fulkerson and Edmonds-Karp algorithms are implemented using something called a **residual network**. This will be explained in more detail on the next pages.

The **residual network** is set up with the **residual capacities** on each edge, where the residual capacity of an edge is the capacity on that edge, minus the flow. So when flow is increased in a edge, the residual capacity is decreased with the same amount.

For each edge in the residual network, there is also a **reversed edge** that points in the opposite direction of the original edge. The residual capacity of a reversed edge is the flow of the original edge. Reversed edges are important for sending flow back on an edge as part of the maximum flow algorithms.



Some of these concepts, like the residual network and the reversed edge, can be hard to understand. That is why these concepts are explained more in detail, and with examples, on the next two pages.

When the maximum flow is found, we get a value for how much flow can be sent through the flow network in total.

## Multiple Source and Sink Vertices

The Ford-Fulkerson and Edmonds-Karp algorithms expects one source vertex and one sink vertex to be able to find the maximum flow.

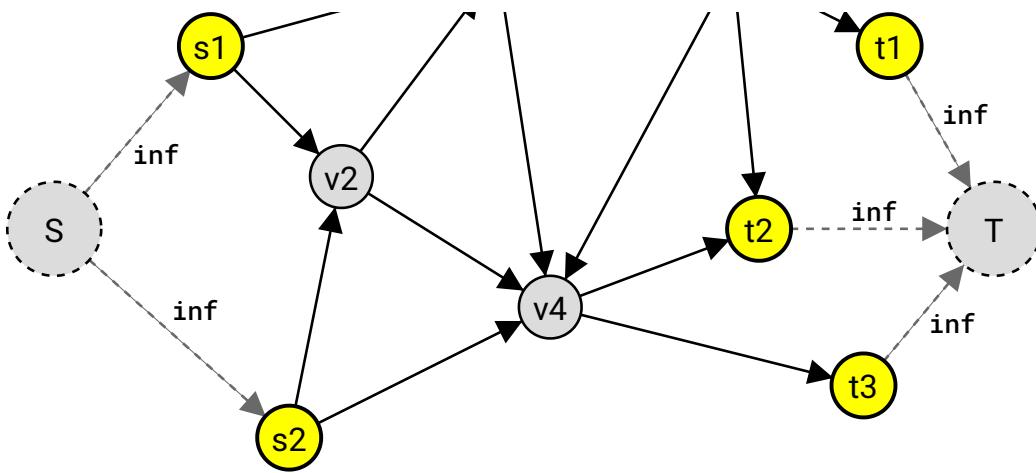
If the graph has more than one source vertex, or more than one sink vertex, the graph should be modified to find the maximum flow.

To modify the graph so that you can run the Ford-Fulkerson or Edmonds-Karp algorithm on it, create an extra super-source vertex if you have multiple source vertices, and create an extra super-sink vertex if you have multiple sink-vertices.

From the super-source vertex, create edges to the original source vertices, with infinite capacities. And create edges from the sink vertices to the super-sink vertex similarly, with infinite capacities.

The image below shows such a graph with two sources  $s_1$  and  $s_2$ , and three sinks  $t_1$ ,  $t_2$ , and  $t_3$ .

To run Ford-Fulkerson or Edmonds-Karp on this graph, a super source  $S$  is created with edges with infinite capacities to the original source nodes, and a super sink  $T$  is created with edges with



The Ford-Fulkerson or Edmonds-Karp algorithm is now able to find maximum flow in a graph with multiple source and sink vertices, by going from the super source  $S$ , to the super sink  $T$ .

## The Max-flow Min-cut Theorem

To understand what this theorem says we first need to know what a cut is.

We create two sets of vertices: one with only the source vertex inside it called "S", and one with all the other vertices inside it (including the sink vertex) called "T".

Now, starting in the source vertex, we can choose to expand set  $S$  by including adjacent vertices, and continue to include adjacent vertices as much as we want as long as we do not include the sink vertex.

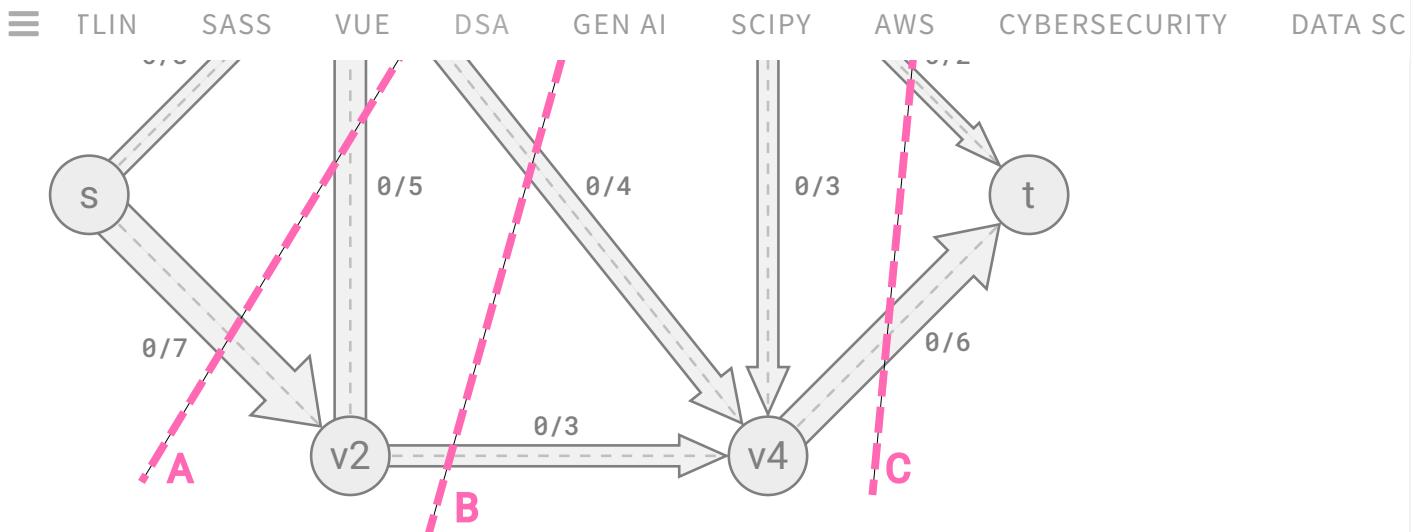
Expanding set  $S$  will shrink set  $T$ , because any vertex belongs either to set  $S$  or set  $T$ .

In such a setup, with any vertex belonging to either set  $S$  or set  $T$ , there is a "cut" between the sets. The cut consists of all the edges stretching from set  $S$  to set  $T$ .

If we add all the capacities from edges going from set  $S$  to set  $T$ , we get the capacity of the cut, which is the total possible flow from source to sink in this cut.

The minimum cut is the cut we can make with the lowest total capacity, which will be the bottleneck.

In the image below, three different cuts are done in the graph from the simulation in the top of this page.



**Cut A:** This cut has vertices  $s$  and  $v_1$  in set S, and the other vertices are in set T. The total capacity of the edges leaving set S in this cut, from sink to source, is  $3+4+7=14$ . We are not adding the capacity from edge  $v_2 \rightarrow v_1$ , because this edge goes in the opposite direction, from sink to source. So the maximum possible flow across cut A is 14.

**Cut B:** The maximum possible flow is  $3+4+3=10$  across cut B.

**Cut C:** The maximum possible flow is  $2+6=8$  across cut C. If we checked all other cuts in the graph, we would not find a cut with a lower total capacity. This is the minimum cut. Have you run the simulation finding the maximum flow in the top of this page? Then you also know that 8 is the maximum flow, which is exactly what the max-flow min-cut theorem says.

The max-flow min-cut theorem says that finding the minimum cut in a graph, is the same as finding the maximum flow, because the value of the minimum cut will be the same value as the maximum flow.

## Practical Implications of The Max-flow Min-cut Theorem

Finding the maximum flow in a graph using an algorithm like Ford-Fulkerson also helps us to understand where the minimum cut is: The minimum cut will be where the edges have reached full capacity.

The minimum cut will be where the bottleneck is, so if we want to increase flow beyond the maximum limit, which is often the case in practical situations, we now know which edges in the graph that needs to be modified to increase the overall flow.

Modifying edges in the minimum cut to allow more flow can be very useful in many situations:

- In logistics, knowing where the bottleneck is, the supply chain can be optimized by changing routes, or increase capacity at critical points, ensuring that goods are moved more effectively from warehouses to consumers.

So using maximum flow algorithms to find the minimum cut, helps us to understand where the system can be modified to allow an even higher throughput.

---

# The Maximum Flow Problem Described Mathematically

The maximum flow problem is not just a topic in Computer Science, it is also a type of Mathematical Optimization, that belongs to the field of Mathematics.

In case you want to understand this better mathematically, the maximum flow problem is described in mathematical terms below.

All edges ( $E$ ) in the graph, going from a vertex ( $u$ ) to a vertex ( $v$ ), have a flow ( $f$ ) that is less than, or equal to, the capacity ( $c$ ) of that edge:

$$\forall (u, v) \in E : f(u, v) \leq c(u, v)$$

This basically just means that the flow in an edge is limited by the capacity in that edge.

Also, for all edges ( $E$ ), a flow in one direction from  $u$  to  $v$  is the same as having a negative flow in the reverse direction, from  $v$  to  $u$ :

$$\forall (u, v) \in E : f(u, v) = -f(v, u)$$

And the expression below states that conservation of flow is kept for all vertices ( $u$ ) except for the source vertex ( $s$ ) and for the sink vertex ( $t$ ):

$$\forall u \in V \setminus \{s, t\} \Rightarrow \sum_{w \in V} f(u, w) = 0$$

This just means that the amount of flow going into a vertex, is the same amount of flow that comes out of that vertex (except for the source and sink vertices).

And at last, all flow leaving the source vertex  $s$ , must end up in the sink vertex  $t$ :

$$\sum_{(s, u) \in E} f(s, u) = \sum_{(v, t) \in E} f(v, t)$$

# DSA Ford-Fulkerson Algorithm

[« Previous](#)[Next »](#)

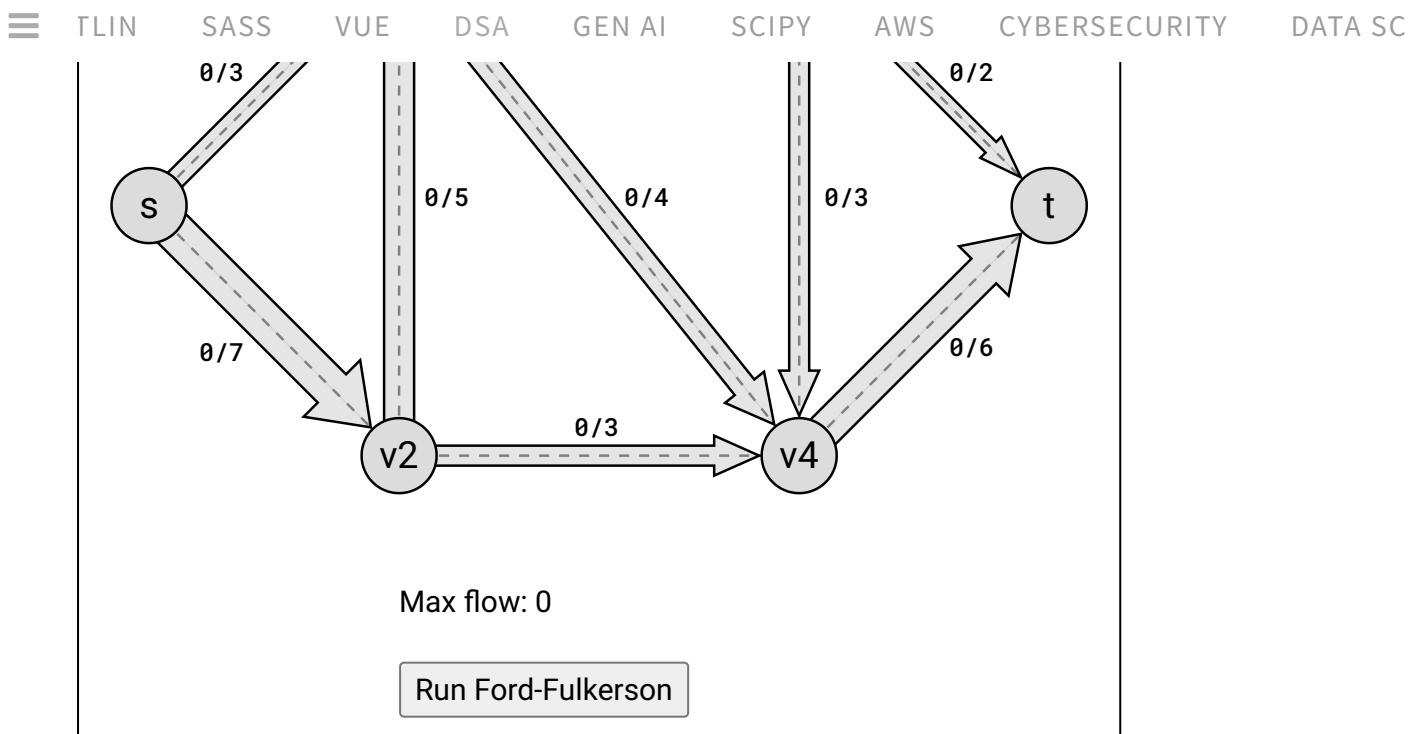
The Ford-Fulkerson algorithm solves the maximum flow problem.

Finding the maximum flow can be helpful in many areas: for optimizing network traffic, for manufacturing, for supply chain and logistics, or for airline scheduling.

## The Ford-Fulkerson Algorithm

The Ford-Fulkerson algorithm solves the maximum flow problem for a directed graph.

The flow comes from a source vertex ( $s$ ) and ends up in a sink vertex ( $t$ ), and each edge in the graph allows a flow, limited by a capacity.



The Ford-Fulkerson algorithm works by looking for a path with available capacity from the source to the sink (called an *augmented path*), and then sends as much flow as possible through that path.

The Ford-Fulkerson algorithm continues to find new paths to send more flow through until the maximum flow is reached.

In the simulation above, the Ford-Fulkerson algorithm solves the maximum flow problem: It finds out how much flow can be sent from the source vertex  $s$ , to the sink vertex  $t$ , and that maximum flow is 8.

The numbers in the simulation above are written in fractions, where the first number is the flow, and the second number is the capacity (maximum possible flow in that edge). So for example,  $0/7$  on edge  $s \rightarrow v_2$ , means there is  $0$  flow, with a capacity of  $7$  on that edge.

**Note:** The Ford-Fulkerson algorithm is often described as a *method* instead of as an *algorithm*, because it does not specify how to find a path where flow can be increased. This means it can be implemented in different ways, resulting in different time complexities. But for this tutorial we will call it an algorithm, and use Depth-First-Search to find the paths.

You can see the basic step-by-step description of how the Ford-Fulkerson algorithm works below, but we need to go into more detail later to actually understand it.

1. Start with zero flow on all edges.
2. Find an *augmented path* where more flow can be sent.
3. Do a *bottleneck calculation* to find out how much flow can be sent through that augmented path.
4. Increase the flow found from the bottleneck calculation for each edge in the augmented path.
5. Repeat steps 2-4 until max flow is found. This happens when a new augmented path can no longer be found.

## Residual Network in Ford-Fulkerson

The Ford-Fulkerson algorithm actually works by creating and using something called a *residual network*, which is a representation of the original graph.

In the residual network, every edge has a *residual capacity*, which is the original capacity of the edge, minus the flow in that edge. The residual capacity can be seen as the leftover capacity in an edge with some flow.

For example, if there is a flow of 2 in the  $v_3 \rightarrow v_4$  edge, and the capacity is 3, the residual flow is 1 in that edge, because there is room for sending 1 more unit of flow through that edge.

## Reversed Edges in Ford-Fulkerson

The Ford-Fulkerson algorithm also uses something called *reversed edges* to send flow back. This is useful to increase the total flow.

For example, the last augmented path  $s \rightarrow v_2 \rightarrow v_4 \rightarrow v_3 \rightarrow t$  in the animation above and in the manual run through below shows how the total flow is increased by one more unit, by actually sending flow back on edge  $v_4 \rightarrow v_3$ , sending the flow in the reverse direction.

Sending flow back in the reverse direction on edge  $v_3 \rightarrow v_4$  in our example means that this 1 unit of flow going out of vertex  $v_3$ , now leaves  $v_3$  on edge  $v_3 \rightarrow t$  instead of  $v_3 \rightarrow v_4$ .

To send flow back, in the opposite direction of the edge, a reverse edge is created for each original edge in the network. The Ford-Fulkerson algorithm can then use these reverse edges to send flow in the reverse direction.

This just means that when there is a flow of 2 on the original edge  $v_3 \rightarrow v_4$ , there is a possibility of sending that same amount of flow back on that edge, but in the reversed direction. Using a reversed edge to push back flow can also be seen as undoing a part of the flow that is already created.

The idea of a residual network with residual capacity on edges, and the idea of reversed edges, are central to how the Ford-Fulkerson algorithm works, and we will go into more detail about this when we implement the algorithm further down on this page.

## Manual Run Through

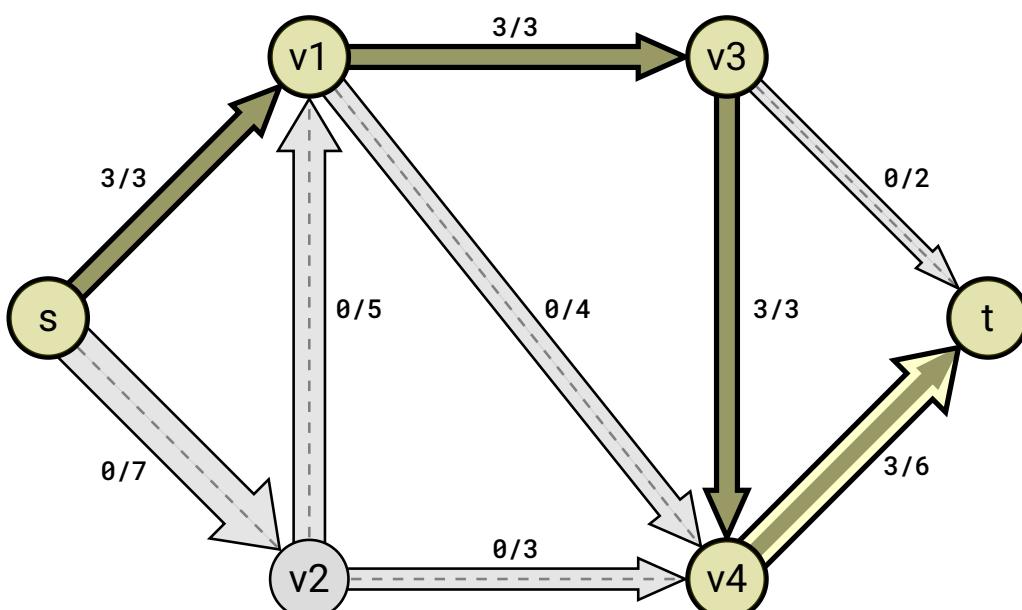
There is no flow in the graph to start with.

To find the maximum flow, the Ford-Fulkerson algorithm must increase flow, but first it needs to find out where the flow can be increased: it must find an augmented path.

The Ford-Fulkerson algorithm actually does not specify how such an augmented path is found (that is why it is often described as a method instead of an algorithm), but we will use Depth First Search (DFS) to find the augmented paths for the Ford-Fulkerson algorithm in this tutorial.

The first augmented path Ford-Fulkerson finds using DFS is  $s \rightarrow v_1 \rightarrow v_3 \rightarrow v_4 \rightarrow t$ .

And using the bottleneck calculation, Ford-Fulkerson finds that 3 is the highest flow that can be sent through the augmented path, so the flow is increased by 3 for all the edges in this path.



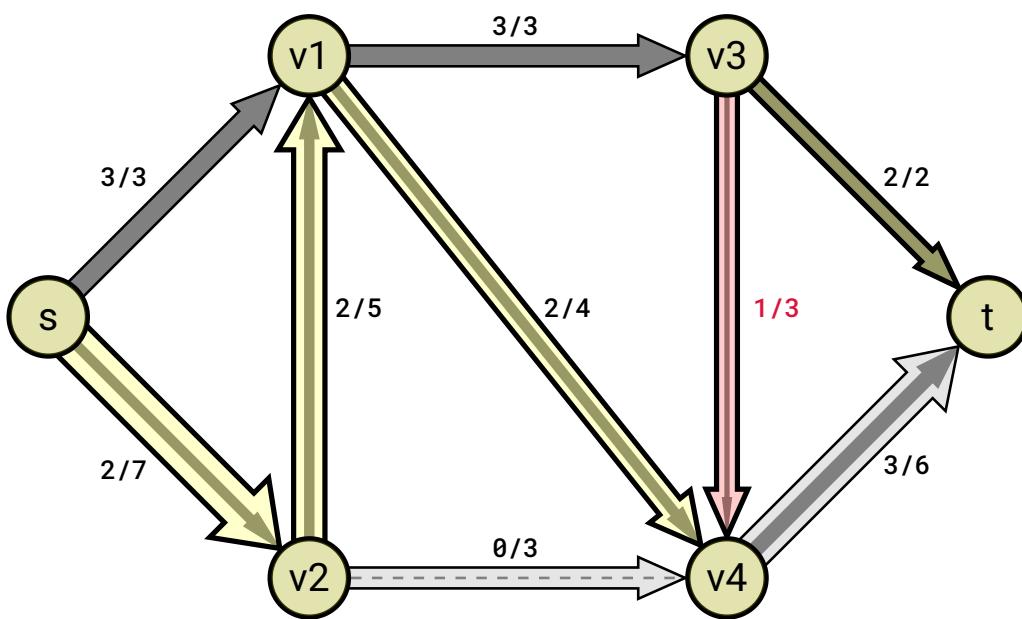
#### 4. Increase the flow along the edges in that path accordingly

The next augmented path is found to be  $s \rightarrow v_2 \rightarrow v_1 \rightarrow v_4 \rightarrow v_3 \rightarrow t$ , which includes the reversed edge  $v_4 \rightarrow v_3$ , where flow is sent back.

The Ford-Fulkerson concept of reversed edges comes in handy because it allows the path finding part of the algorithm to find an augmented path where reversed edges can also be included.

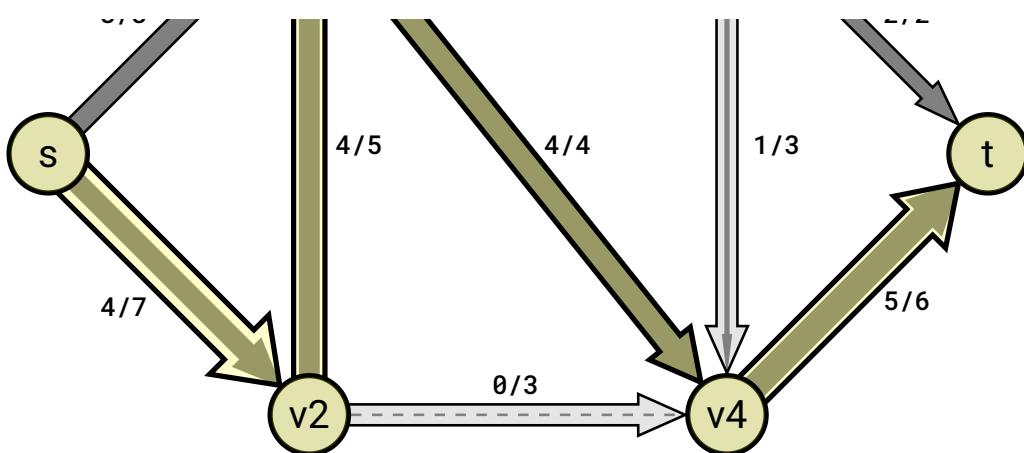
In this specific case that means that a flow of 2 can be sent back on edge  $v_3 \rightarrow v_4$ , going into  $v_3 \rightarrow t$  instead.

The flow can only be increased by 2 in this path because that is the capacity in the  $v_3 \rightarrow t$  edge.



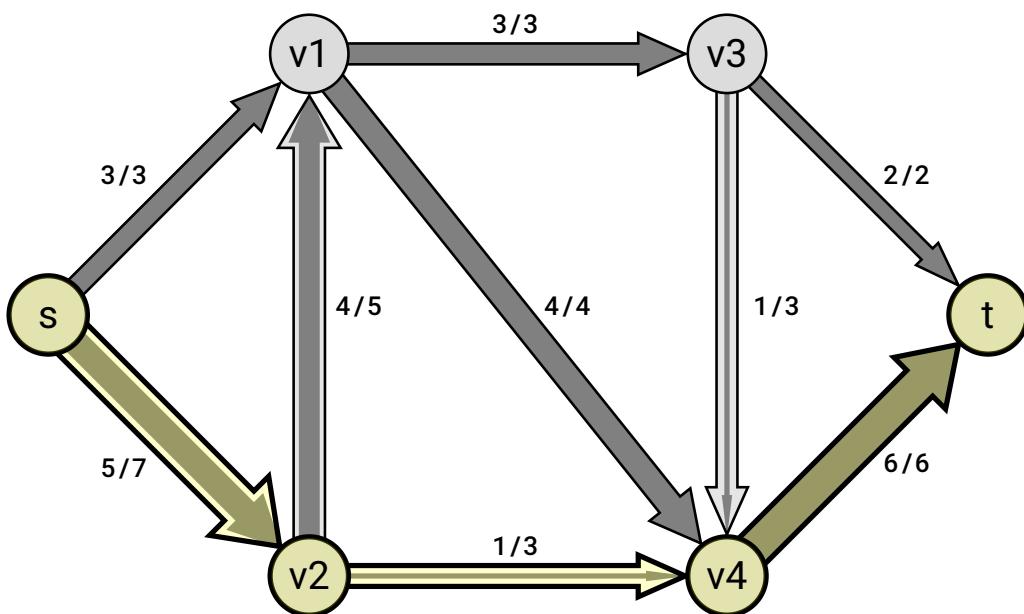
The next augmented path is found to be  $s \rightarrow v_2 \rightarrow v_1 \rightarrow v_4 \rightarrow t$ .

The flow can be increased by 2 in this path. The bottleneck (limiting edge) is  $v_1 \rightarrow v_4$  because there is only room for sending two more units of flow in that edge.



The next and last augmenting path is  $s \rightarrow v_2 \rightarrow v_4 \rightarrow t$ .

The flow can only be increased by 1 in this path because of edge  $v_4 \rightarrow t$  being the bottleneck in this path with only space for one more unit of flow ( $\text{capacity} - \text{flow} = 1$ ).



At this point, a new augmenting path cannot be found (it is not possible to find a path where more flow can be sent through from  $s$  to  $t$ ), which means the max flow has been found, and the Ford-Fulkerson algorithm is finished.

The maximum flow is 8. As you can see in the image above, the flow (8) is the same going out of the source vertex  $s$ , as the flow going into the sink vertex  $t$ .

Also, if you take any other vertex than  $s$  or  $t$ , you can see that the amount of flow going into a vertex, is the same as the flow going out of it. This is what we call *conservation of flow*, and this must hold for all such flow networks (directed graphs where each edge has a flow and a capacity).

To implement the Ford-Fulkerson algorithm, we create a `Graph` class. The `Graph` represents the graph with its vertices and edges:

```
1 | class Graph:  
2 |     def __init__(self, size):
```

```
6 |
```

```
9 |
```

**Line 3:** We create the `adj_matrix` to hold all the edges and edge capacities. Initial values are set to `0`.

**Line 4:** `size` is the number of vertices in the graph.

**Line 5:** The `vertex_data` holds the names of all the vertices.

**Line 7-8:** The `add_edge` method is used to add an edge from vertex `u` to vertex `v`, with capacity `c`.

**Line 10-12:** The `add_vertex_data` method is used to add a vertex name to the graph. The index of the vertex is given with the `vertex` argument, and `data` is the name of the vertex.

The `Graph` class also contains the `dfs` method to find augmented paths, using Depth-First-Search:

```
def dfs(self, s, t, visited=None, path=None):  
    if visited is None:  
        visited = [False] * self.size  
    if path is None:
```

22 |

25 |

31 |

**Line 15-18:** The `visited` array helps to avoid revisiting the same vertices during the search for an augmented path. Vertices that belong to the augmented path are stored in the `path` array.

**Line 20-21:** The current vertex is marked as visited, and then added to the path.

**Line 23-24:** If the current vertex is the sink node, we have found an augmented path from the source vertex to the sink vertex, so that path can be returned.

**Line 26-30:** Looping through all edges in the adjacency matrix starting from the current vertex `s`, `ind` represents an adjacent node, and `val` is the residual capacity on the edge to that vertex. If the adjacent vertex is not visited, and has residual capacity on the edge to it, go to that node and continue searching for a path from that vertex.

**Line 32:** `None` is returned if no path is found.

The `fordFulkerson` method is the last method we add to the `Graph` class:

```
def fordFulkerson(self, source, sink):
    max_flow = 0

    path = self.dfs(source, sink)
    while path:
        path_flow = float("Inf")
        for i in range(len(path) - 1):
            u, v = path[i], path[i + 1]
            path_flow = min(path_flow, self.adj_matrix[u][v])
```

48 |

50 |

53 |

54 |

55 |

56 |

```
    path = self.dfs(source, sink)
    return max_flow
```

Initially, the `max_flow` is `0`, and the `while` loop keeps increasing the `max_flow` as long as there is an augmented path to increase flow in.

**Line 37:** The augmented path is found.

**Line 39-42:** Every edge in the augmented path is checked to find out how much flow can be sent through that path.

**Line 44-46:** The residual capacity (capacity minus flow) for every forward edge is reduced as a result of increased flow.

**Line 47:** This represents the reversed edge, used by the Ford-Fulkerson algorithm so that flow can be sent back (undone) on the original forward edges. It is important to understand that these reversed edges are not in the original graph, they are fictitious edges introduced by Ford-Fulkerson to make the algorithm work.

**Line 49:** Every time flow is increased over an augmented path, `max_flow` is increased by the same value.

**Line 51-52:** This is just for printing the augmented path before the algorithm starts the next iteration.

After defining the `Graph` class, the vertices and edges must be defined to initialize the specific graph, and the complete code for the Ford-Fulkerson algorithm example looks like this:

## Example

Python:

≡ TLIN

SASS

VUE

DSA

GEN AI

SCIPY

AWS

CYBERSECURITY

DATA SC

```
4         self.size = size
5         self.vertex_data = [''] * size
6
7     def add_edge(self, u, v, c):
8         self.adj_matrix[u][v] = c
9
10    def add_vertex_data(self, vertex, data):
11        if 0 <= vertex < self.size:
12            self.vertex_data[vertex] = data
13
14    def dfs(self, s, t, visited=None, path=None):
15        if visited is None:
16            visited = [False] * self.size
17        if path is None:
```

[Run Example »](#)

# Time Complexity for The Ford-Fulkerson Algorithm

The time complexity for the Ford-Fulkerson varies with the number of vertices  $V$ , the number of edges  $E$ , and it actually varies with the maximum flow  $f$  in the graph as well.

The reason why the time complexity varies with the maximum flow  $f$  in the graph, is because in a graph with a high throughput, there will be more augmented paths that increase flow, and that means the DFS method that finds these augmented paths will have to run more times.

Depth-first search (DFS) has time complexity  $O(V + E)$ .

DFS runs once for every new augmented path. If we assume that each augmented graph increase flow by 1 unit, DFS must run  $f$  times, as many times as the value of maximum flow.

This means that time complexity for the Ford-Fulkerson algorithm, using DFS, is

$$O((V + E) \cdot f)$$

For *dense graphs*, where  $E > V$ , time complexity for DFS can be simplified to  $O(E)$ , which means that the time complexity for the Ford-Fulkerson algorithm also can be simplified to

The next algorithm we will describe that finds maximum flow is [the Edmonds-Karp algorithm](#).

The Edmonds-Karp algorithm is very similar to Ford-Fulkerson, but it uses BFS instead of DFS to find augmented paths, which leads to fewer iterations to find maximum flow.

[« Previous](#)[Next »](#)

### W3schools Pathfinder

Track your progress - it's free!

[Sign Up](#) [Log in](#)

ADVERTISEMENT

Get Lifelong Access to  
**All Current & Future Courses**

Our certifications are recognised worldwide.

Original price: \$1,465

**Discount price: \$695**

**Start Today!**



### COLOR PICKER



# DSA Edmonds-Karp Algorithm

[« Previous](#)[Next »](#)

The Edmonds-Karp algorithm solves the maximum flow problem.

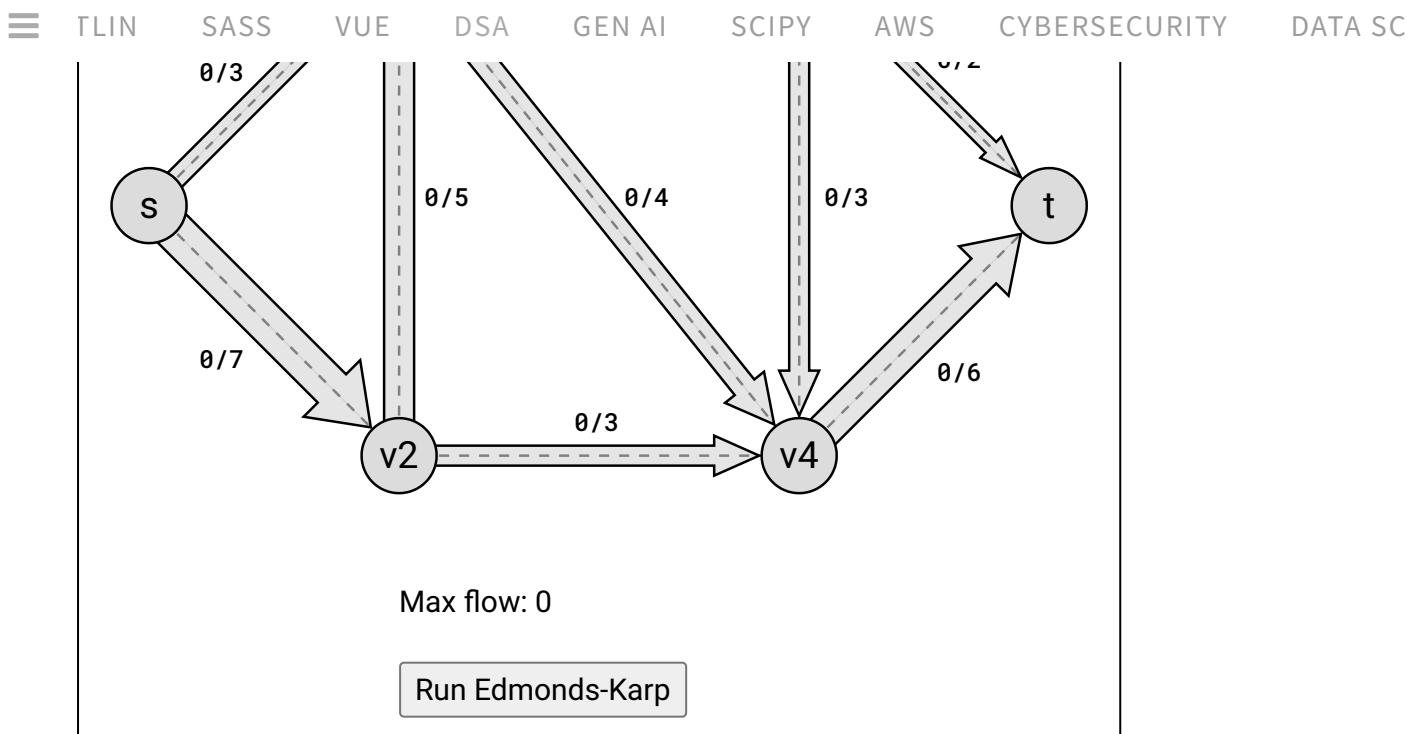
Finding the maximum flow can be helpful in many areas: for optimizing network traffic, for manufacturing, for supply chain and logistics, or for airline scheduling.

## The Edmonds-Karp Algorithm

The Edmonds-Karp algorithm solves the maximum flow problem for a directed graph.

The flow comes from a source vertex ( $s$ ) and ends up in a sink vertex ( $t$ ), and each edge in the graph allows a flow, limited by a capacity.

The Edmonds-Karp algorithm is very similar to the Ford-Fulkerson algorithm, except the Edmonds-Karp algorithm uses Breadth First Search (BFS) to find augmented paths to increase flow.



The Edmonds-Karp algorithm works by using Breadth-First Search (BFS) to find a path with available capacity from the source to the sink (called an *augmented path*), and then sends as much flow as possible through that path.

The Edmonds-Karp algorithm continues to find new paths to send more flow through until the maximum flow is reached.

In the simulation above, the Edmonds-Karp algorithm solves the maximum flow problem: It finds out how much flow can be sent from the source vertex  $s$ , to the sink vertex  $t$ , and that maximum flow is 8.

The numbers in the simulation above are written in fractions, where the first number is the flow, and the second number is the capacity (maximum possible flow in that edge). So for example,  $0/7$  on edge  $s \rightarrow v_2$ , means there is  $0$  flow, with a capacity of  $7$  on that edge.

You can see the basic step-by-step description of how the Edmonds-Karp algorithm works below, but we need to go into more detail later to actually understand it.

### How it works:

1. Start with zero flow on all edges.
2. Use BFS to find an *augmented path* where more flow can be sent.
3. Do a *bottleneck calculation* to find out how much flow can be sent through that augmented path.

# Residual Network in Edmonds-Karp

The Edmonds-Karp algorithm works by creating and using something called a *residual network*, which is a representation of the original graph.

In the residual network, every edge has a *residual capacity*, which is the original capacity of the edge, minus the flow in that edge. The residual capacity can be seen as the leftover capacity in an edge with some flow.

For example, if there is a flow of 2 in the  $v_3 \rightarrow v_4$  edge, and the capacity is 3, the residual flow is 1 in that edge, because there is room for sending 1 more unit of flow through that edge.

# Reversed Edges in Edmonds-Karp

The Edmonds-Karp algorithm also uses something called *reversed edges* to send flow back. This is useful to increase the total flow.

To send flow back, in the opposite direction of the edge, a reverse edge is created for each original edge in the network. The Edmonds-Karp algorithm can then use these reverse edges to send flow in the reverse direction.

A reversed edge has no flow or capacity, just residual capacity. The residual capacity for a reversed edge is always the same as the flow in the corresponding original edge.

In our example, the edge  $v_1 \rightarrow v_3$  has a flow of 2, which means there is a residual capacity of 2 on the corresponding reversed edge  $v_3 \rightarrow v_1$ .

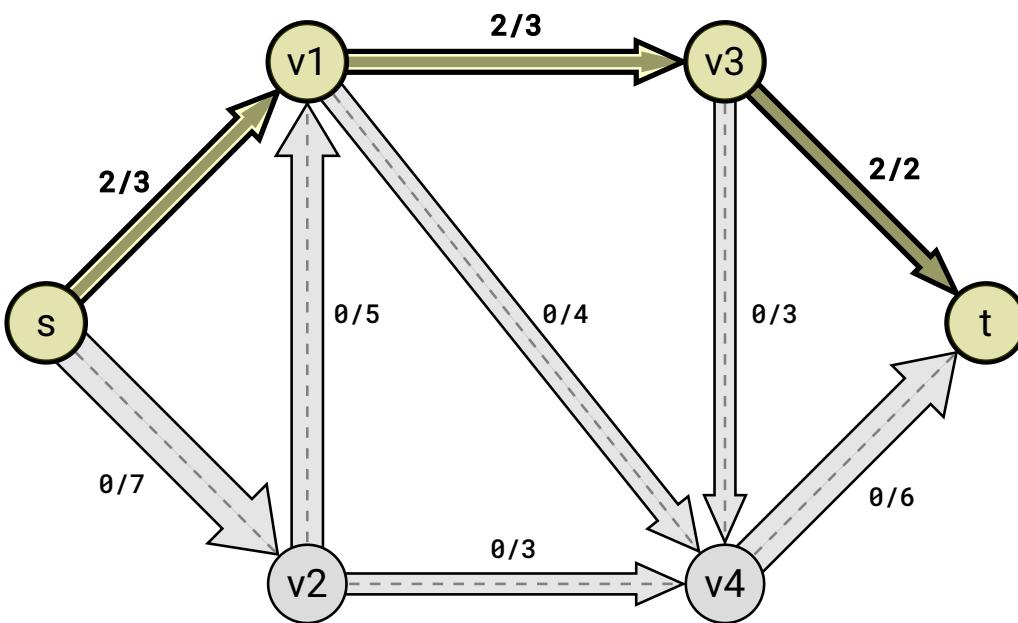
This just means that when there is a flow of 2 on the original edge  $v_1 \rightarrow v_3$ , there is a possibility of sending that same amount of flow back on that edge, but in the reversed direction. Using a reversed edge to push back flow can also be seen as undoing a part of the flow that is already created.

The idea of a residual network with residual capacity on edges, and the idea of reversed edges, are central to how the Edmonds-Karp algorithm works, and we will go into more detail about this when we implement the algorithm further down on this page.

The Edmonds-Karp algorithm starts with using Breadth-First Search to find an augmented path where flow can be increased, which is  $s \rightarrow v_1 \rightarrow v_3 \rightarrow t$ .

After finding the augmented path, a bottleneck calculation is done to find how much flow can be sent through that path, and that flow is: 2.

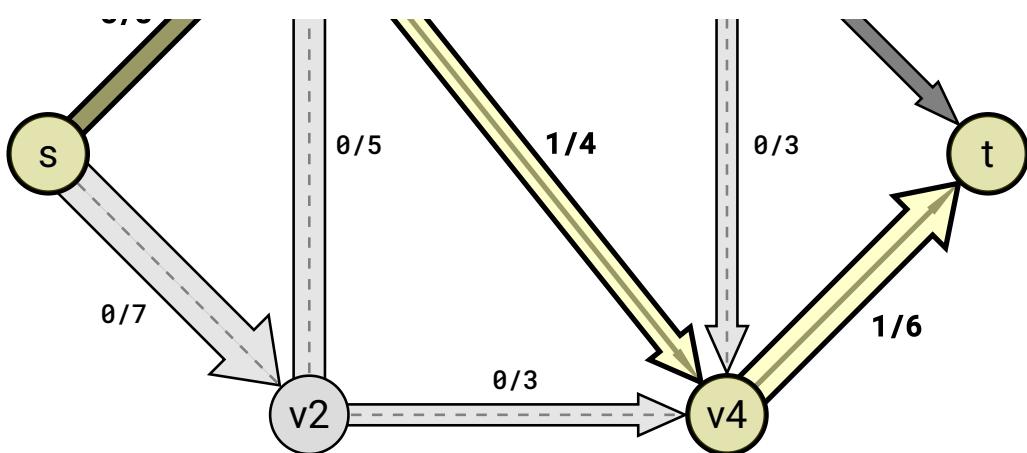
So a flow of 2 is sent over each edge in the augmented path.



The next iteration of the Edmonds-Karp algorithm is to do these steps again: Find a new augmented path, find how much the flow in that path can be increased, and increase the flow along the edges in that path accordingly.

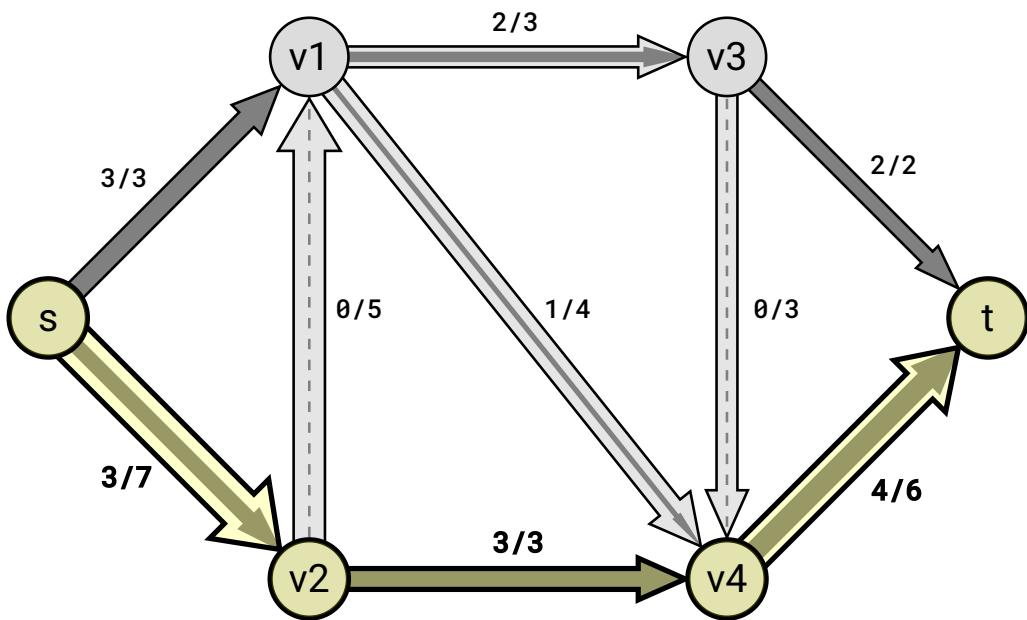
The next augmented path is found to be  $s \rightarrow v_1 \rightarrow v_4 \rightarrow t$ .

The flow can only be increased by 1 in this path because there is only room for one more unit of flow in the  $s \rightarrow v_1$  edge.



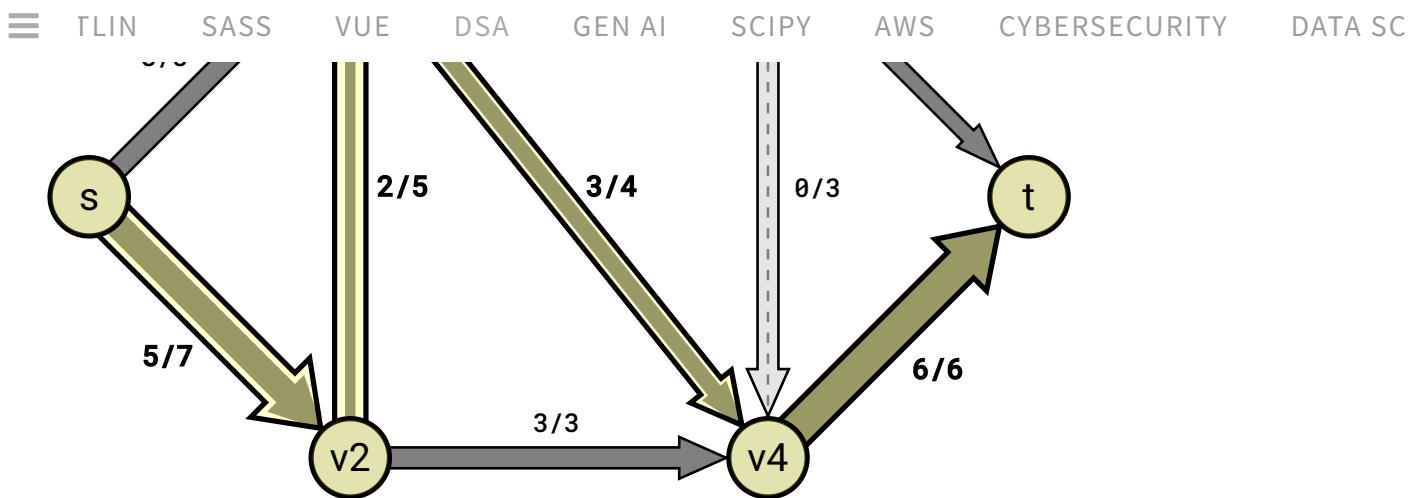
The next augmented path is found to be  $s \rightarrow v_2 \rightarrow v_4 \rightarrow t$ .

The flow can be increased by 3 in this path. The bottleneck (limiting edge) is  $v_2 \rightarrow v_4$  because the capacity is 3.



The last augmented path found is  $s \rightarrow v_2 \rightarrow v_1 \rightarrow v_4 \rightarrow t$ .

The flow can only be increased by 2 in this path because of edge  $v_4 \rightarrow t$  being the bottleneck in this path with only space for 2 more units of flow ( $capacity - flow = 1$ ).



At this point, a new augmenting path cannot be found (it is not possible to find a path where more flow can be sent through from  $s$  to  $t$ ), which means the max flow has been found, and the Edmonds-Karp algorithm is finished.

The maximum flow is 8. As you can see in the image above, the flow (8) is the same going out of the source vertex  $s$ , as the flow going into the sink vertex  $t$ .

Also, if you take any other vertex than  $s$  or  $t$ , you can see that the amount of flow going into a vertex, is the same as the flow going out of it. This is what we call *conservation of flow*, and this must hold for all such flow networks (directed graphs where each edge has a flow and a capacity).

## Implementation of The Edmonds-Karp Algorithm

To implement the Edmonds-Karp algorithm, we create a `Graph` class.

The `Graph` represents the graph with its vertices and edges:

```
1 | class Graph:  
2 |     def __init__(self, size):
```

```
6 |
```

```
9 |
```

**Line 3:** We create the `adj_matrix` to hold all the edges and edge capacities. Initial values are set to `0`.

**Line 4:** `size` is the number of vertices in the graph.

**Line 5:** The `vertex_data` holds the names of all the vertices.

**Line 7-8:** The `add_edge` method is used to add an edge from vertex `u` to vertex `v`, with capacity `c`.

**Line 10-12:** The `add_vertex_data` method is used to add a vertex name to the graph. The index of the vertex is given with the `vertex` argument, and `data` is the name of the vertex.

The `Graph` class also contains the `bfs` method to find augmented paths, using Breadth-First-Search:

```
14 |     def bfs(self, s, t, parent):
```

```
19 |
```

```
22 |
```

```
28 |
```

**Line 15-18:** The `visited` array helps to avoid revisiting the same vertices during the search for an augmented path. The `queue` holds vertices to be explored, the search always starts with the source vertex `s`.

**Line 24-27:** If the adjacent vertex is not visited yet, and there is a residual capacity on the edge to that vertex: add it to the queue of vertices that needs to be explored, mark it as visited, and set the `parent` of the adjacent vertex to be the current vertex `u`.

The `parent` array holds the parent of a vertex, creating a path from the sink vertex, backwards to the source vertex. The `parent` is used later in the Edmonds-Karp algorithm, outside the `bfs` method, to increase flow in the augmented path.

**Line 29:** The last line returns `visited[t]`, which is `true` if the augmented path ends in the sink node `t`. Returning `true` means that an augmenting path has been found.

The `edmonds_karp` method is the last method we add to the `Graph` class:

```
def edmonds_karp(self, source, sink):
    parent = [-1] * self.size
    max_flow = 0

    while self.bfs(source, sink, parent):
        path_flow = float("Inf")
        s = sink
        while(s != source):
            path_flow = min(path_flow, self.adj_matrix[parent[s]][s])
            s = parent[s]

        max_flow += path_flow
        v = sink
        while(v != source):
            u = parent[v]
            self.adj_matrix[u][v] -= path_flow
            self.adj_matrix[v][u] += path_flow
            v = parent[v]

        path = []
        v = sink
        while(v != source):
            path.append(v)
            v = parent[v]
        path.append(source)
        path.reverse()
        path_names = [self.vertex_data[node] for node in path]
```

Initially, the `parent` array holds invalid index values, because there is no augmented path to begin with, and the `max_flow` is `0`, and the `while` loop keeps increasing the `max_flow` as long as there is an augmented path to increase flow in.

**Line 35:** The outer `while` loop makes sure the Edmonds-Karp algorithm keeps increasing flow as long as there are augmented paths to increase flow along.

**Line 36-37:** The initial flow along an augmented path is infinite, and the possible flow increase will be calculated starting with the sink vertex.

**Line 38-40:** The value for `path_flow` is found by going backwards from the sink vertex towards the source vertex. The lowest value of residual capacity along the path is what decides how much flow can be sent on the path.

**Line 42:** `path_flow` is increased by the `path_flow`.

**Line 44-48:** Stepping through the augmented path, going backwards from sink to source, the residual capacity is decreased with the `path_flow` on the forward edges, and the residual capacity is increased with the `path_flow` on the reversed edges.

**Line 50-58:** This part of the code is just for printing so that we are able to track each time an augmented path is found, and how much flow is sent through that path.

After defining the `Graph` class, the vertices and edges must be defined to initialize the specific graph, and the complete code for the Edmonds-Karp algorithm example looks like this:

## Example

Python:

```
class Graph:
    def __init__(self, size):
        self.adj_matrix = [[0] * size for _ in range(size)]
        self.size = size
        self.vertex_data = [''] * size

    def add_edge(self, u, v, c):
        self.adj_matrix[u][v] = c
```

```
15     def bfs(self, s, t, parent):  
16         visited = [False] * self.size  
17         queue = [] # Using list as a queue
```

[Run Example »](#)

# Time Complexity for The Edmonds-Karp Algorithm

The difference between Edmonds-Karp and Ford-Fulkerson is that Edmonds-Karp uses Breadth-First Search (BFS) to find augmented paths, while Ford-Fulkerson uses Depth-First Search (DFS).

This means that the time it takes to run Edmonds-Karp is easier to predict than Ford-Fulkerson, because Edmonds-Karp is not affected by the maximum flow value.

With the number of vertices  $V$ , the number of edges  $E$ , the time complexity for the Edmonds-Karp algorithm is

$$O(V \cdot E^2)$$

This means Edmonds-Karp does not depend on the maximum flow, like Ford-Fulkerson does, but on how many vertices and edges we have.

The reason we get this time complexity for Edmonds-Karp is that it runs BFS which has time complexity  $O(E + V)$ .

But if we assume a bad case scenario for Edmonds-Karp, with a dense graph, where the number of edges  $E$  is much greater than the number of vertices  $V$ , time complexity for BFS becomes  $O(E)$ .

BFS must run one time for every augmented path, and there can actually be found close to  $V \cdot E$  augmented paths during running of the Edmonds-Karp algorithm.

So, BFS with time complexity  $O(E)$  can run close to  $V \cdot E$  times in the worst case, which means we get a total time complexity for Edmonds-Karp:  $O(V \cdot E \cdot E) = O(V \cdot E^2)$ .

# DSA Time Complexity

[« Previous](#)[Next »](#)

## Runtime

To fully understand algorithms we must understand how to evaluate the time an algorithm needs to do its job, the runtime.

Exploring the runtime of algorithms is important because using an inefficient algorithm could make our program slow or even unworkable.

By understanding algorithm runtime we can choose the right algorithm for our need, and we can make our programs run faster and handle larger amounts of data effectively.

## Actual Runtime

When considering the runtime for different algorithms, we will **not** look at the actual time an implemented algorithm uses to run, and here is why.

If we implement an algorithm in a programming language, and run that program, the actual time it will use depends on many factors:

- the programming language used to implement the algorithm
- how the programmer writes the program for the algorithm
- the compiler or interpreter used so that the implemented algorithm can run
- the hardware on the computer the algorithm is running on
- the operating system and other tasks going on on the computer
- the amount of data the algorithm is working on

With all these different factors playing a part in the actual runtime for an algorithm, how can we know if one algorithm is faster than another? We need to find a better measure of runtime.

To evaluate and compare different algorithms, instead of looking at the actual runtime for an algorithm, it makes more sense to use something called time complexity.

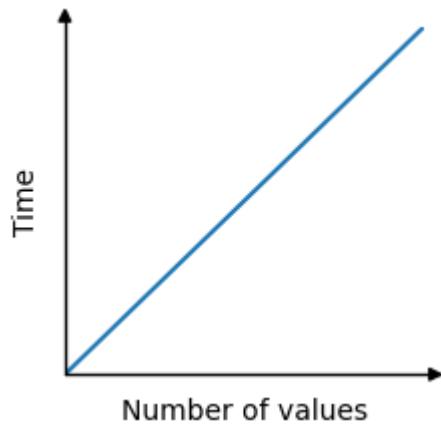
Time complexity is more abstract than actual runtime, and does not consider factors such as programming language or hardware.

Time complexity is the number of operations needed to run an algorithm on large amounts of data. And the number of operations can be considered as time because the computer uses some time for each operation.

For example, in the algorithm that finds the lowest value in an array, each value in the array must be compared one time. Every such comparison can be considered an operation, and each operation takes a certain amount of time. So the total time the algorithm needs to find the lowest value depends on the number of values in the array.

The time it takes to find the lowest value is therefore linear with the number of values. 100 values results in 100 comparisons, and 5000 values results in 5000 comparisons.

The relationship between time and the number of values in the array is linear, and can be displayed in a graph like this:



## "One Operation"

When talking about "operations" here, "one operation" might take one or several CPU cycles, and it really is just a word helping us to abstract, so that we can understand what time complexity is, and so that we can find the time complexity for different algorithms.

*One operation in an algorithm can be understood as something we do in each iteration of the algorithm, or for each piece of data, that takes constant time.*

We say that an operation takes "constant time" if it takes the same time regardless of the amount of data ( $n$ ) the algorithm is processing. Comparing two specific array elements, and swapping them if one is bigger than the other, takes the same time if the array contains 10 or 1000 elements.

## Big O Notation

In mathematics, Big O notation is used to describe the upper bound of a function.

In computer science, Big O notation is used more specifically to find the worst case time complexity for an algorithm.

Big O notation uses a capital letter O with parenthesis  $O()$ , and inside the parenthesis there is an expression that indicates the algorithm runtime. Runtime is usually expressed using  $n$ , which is the number of values in the data set the algorithm is working on.

Below are some examples of Big O notation for different algorithms, just to get the idea:

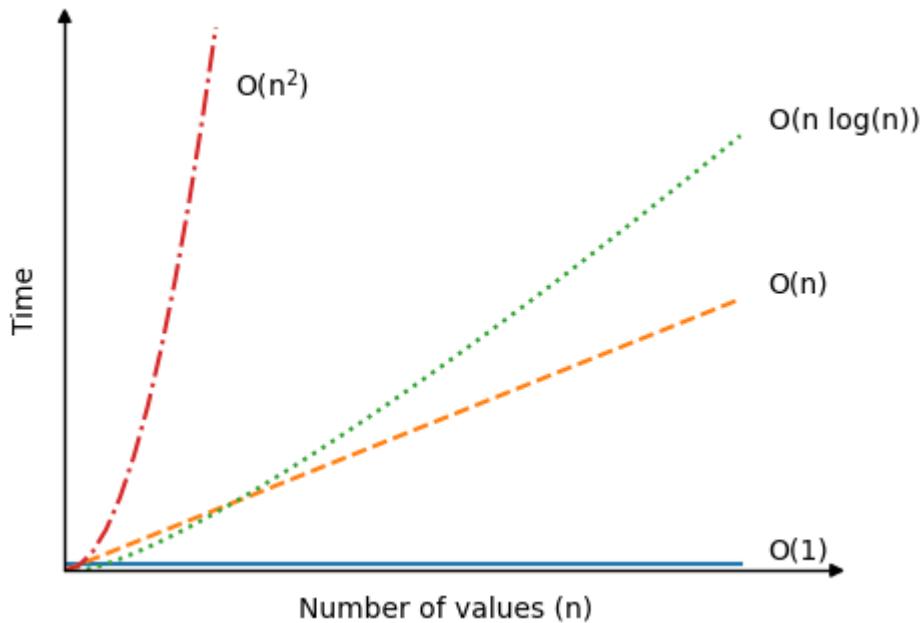
Time Complexity	Algorithm
$O(1)$	Looking up a specific element in an array, like this for example:  <code>print( my_array[97] )</code>  No matter the size of the array, an element can be looked up directly, it just requires one operation. (This is not really an algorithm by the way, but it can help us to understand how time complexity works.)
$O(n)$	<u>Finding the lowest value</u> . The algorithm must do $n$ operations in an array with $n$ values to find the lowest value, because the algorithm must compare each value one time.

Large data sets slows down these algorithms significantly. With just an increase in  $n$  from 100 to 200 values, the number of operations can increase by as much as 30000!

$O(n \log n)$

The [Quicksort algorithm](#) is faster on average than the three sorting algorithms mentioned above, with  $O(n \log n)$  being the average and not the worst case time. Worst case time for Quicksort is also  $O(n^2)$ , but it is the average time that makes Quicksort so interesting. We will learn about Quicksort later.

Here is how time increases when the number of values  $n$  increase for different algorithms:



## Best, Average and Worst Case

'Worst case' time complexity has already been mentioned when explaining Big O notation, but how can an algorithm have a worst case scenario?

The algorithm that finds the lowest value in an array with  $n$  values requires  $n$  operations to do so, and that is always the same. So this algorithm has the same best, average, and worst case scenarios.

runtimes, depending on the values it is sorting.

Just imagine you have to sort 20 values manually from lowest to highest:

8, 16, 19, 15, 2, 17, 4, 11, 6, 1, 7, 13, 5, 3, 9, 12, 14, 20, 18, 10

This will take you some seconds to finish.

Now, imagine you have to sort 20 values that are almost sorted already:

1, 2, 3, 4, 5, 20, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19

You can sort the values really fast, by just moving 20 to the end of the list and you are done, right?

Algorithms work similarly: For the same amount of data they can sometimes be slow and sometimes fast. So to be able to compare different algorithms' time complexities, we usually look at the worst-case scenario using Big O notation.

---

## Big O Explained Mathematically

*Depending on your background in Mathematics, this section might be hard to understand. It is meant to create a more solid mathematical basis for those who need Big O explained more thoroughly.*

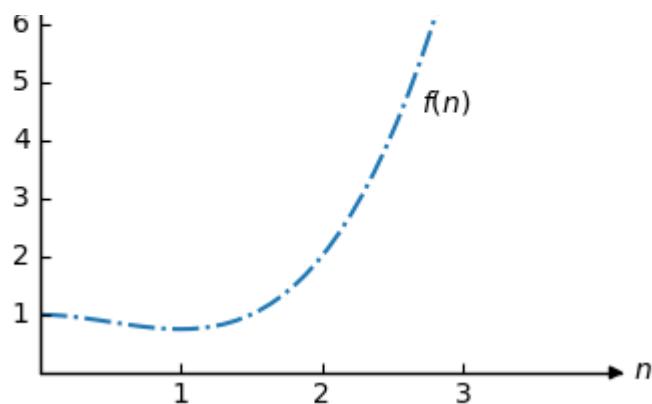
*If you do not understand this now, don't worry, you can come back later. And if the math here is way over your head, don't worry too much about it, you can still enjoy the different algorithms in this tutorial, learn how to program them, and understand how fast or slow they are.*

In Mathematics, Big O notation is used to create an upper bound for a function, and in Computer Science, Big O notation is used to describe how the runtime of an algorithm increases when the number of data values  $n$  increase.

For example, consider the function:

$$f(n) = 0.5n^3 - 0.75n^2 + 1$$

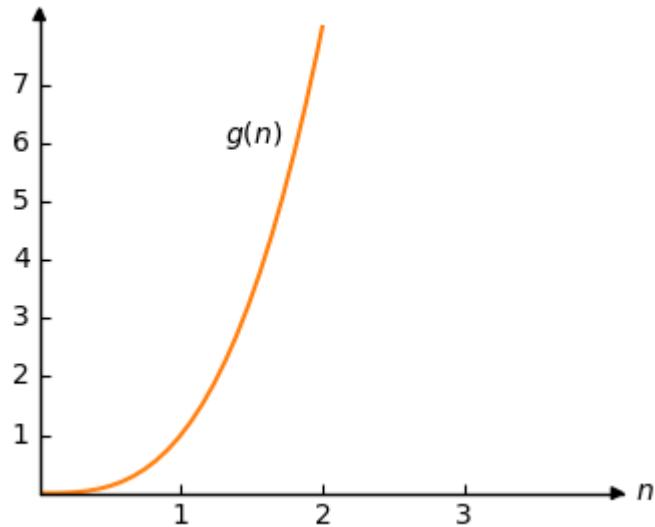
The graph for the function  $f$  looks like this:



Consider another function:

$$g(n) = n^3$$

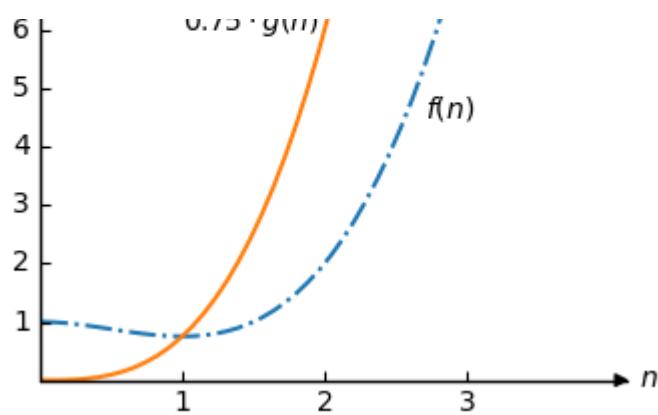
Which we can draw like this:



Using Big O notation we can say that  $O(g(n))$  is an upper bound for  $f(n)$  because we can choose a constant  $C$  so that  $C \cdot g(n) > f(n)$  as long as  $n$  is big enough.

Ok, let's try. We choose  $C = 0.75$  so that  $C \cdot g(n) = 0.75 \cdot n^3$ .

Now let's draw  $0.75 \cdot g(n)$  and  $f(n)$  in the same plot:



We can see that  $O(g(n)) = O(n^3)$  is the upper bound for  $f(n)$  because  $0.75 \cdot g(n) > f(n)$  for all  $n$  larger than 1.

In the example above  $n$  must be larger than 1 for  $O(n^3)$  to be an upper bound. We call this limit  $n_0$ .

## Definition

Let  $f(n)$  and  $g(n)$  be two functions. We say that  $f(n)$  is  $O(g(n))$  if and only if there are positive constants  $C$  and  $n_0$  such that

$$C \cdot g(n) > f(n)$$

for all  $n > n_0$ .

When evaluating time complexity for an algorithm, it is ok that  $O()$  is only true for a large number of values  $n$ , because that is when time complexity becomes important. To put it differently: if we are sorting, 10, 20 or 100 values, the time complexity for the algorithm is not so interesting, because the computer will sort the values in a short time anyway.

[« Previous](#)[Next »](#)

# DSA Bubble Sort Time Complexity

[« Previous](#)[Next »](#)

See [the previous page](#) for a general explanation of what time complexity is.

## Bubble Sort Time Complexity

The [Bubble Sort algorithm](#) goes through an array of  $n$  values  $n - 1$  times in a worst case scenario.

The first time the algorithm runs through the array, every value is compared to the next, and swaps the values if the left value is larger than the right. This means that the highest value bubbles up, and the unsorted part of the array becomes shorter and shorter until the sorting is done. So on average,  $\frac{n}{2}$  elements are considered when the algorithm goes through the array comparing and swapping values.

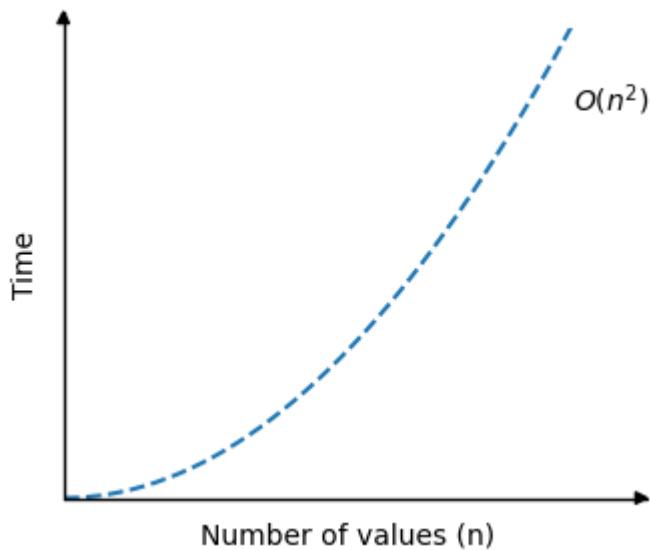
We can start calculating the number of operations done by the Bubble Sort algorithm on  $n$  values:

$$\text{Operations} = (n - 1) \cdot \frac{n}{2} = \frac{n^2}{2} - \frac{n}{2}$$

When looking at the time complexity for algorithms, we look at very large data sets, meaning  $n$  is a very big number. And for a very big number  $n$ , the term  $\frac{n^2}{2}$  becomes a lot bigger than the term  $\frac{n}{2}$ . So large in fact, that we can approximate by simply removing that second term  $\frac{n}{2}$ .

$$\text{Operations} = \frac{n^2}{2} - \frac{n}{2} \approx \frac{n^2}{2} = \frac{1}{2} \cdot n^2$$

When we are looking at time complexity like we are here, using Big O notation, factors are disregarded, so factor  $\frac{1}{2}$  is omitted. This means that the run time for the Bubble Sort algorithm can be described with time complexity, using Big O notation like this:



As you can see, the run time increases really fast when the size of the array is increased.

Luckily there are sorting algorithms that are faster than this, like [Quicksort](#).

## Bubble Sort Simulation

Choose the number of values in an array, and run this simulation to see how the number of operations Bubble Sort needs on an array of  $n$  elements is  $O(n^2)$ :

Set values:  300

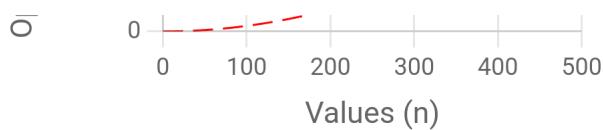
Random

Worst Case

Best Case

10 Random

Operations: 0



The red line above represents the upper bound time complexity  $O(n^2)$ , and the actual function in this case is  $1.05 \cdot n^2$ .

A function  $f(n)$  is said to be  $O(g(n))$  if we have a positive constant  $C$  so that  $C \cdot g(n) > f(n)$  for a large number of values  $n$ .

In this case  $f(n)$  is the number of operations used by Bubble Sort,  $g(n) = n^2$  and  $C = 1.05$ .

Read more about Big O notation and time complexity on [this page](#).

[« Previous](#)[Next »](#)

**W3schools Pathfinder**  
**Track your progress - it's free!**

[Sign Up](#) [Log in](#)

ADVERTISEMENT

# DSA Selection Sort Time Complexity

[« Previous](#)[Next »](#)

See [this page](#) for a general explanation of what time complexity is.

## Selection Sort Time Complexity

The [Selection Sort algorithm](#) goes through all elements in an array, finds the lowest value, and moves it to the front of the array, and does this over and over until the array is sorted.

Selection Sort goes through an array of  $n$  values  $n - 1$  times. This is because when the algorithm has sorted all values except the last, the last value must also be in its correct place.

The first time the algorithm runs through the array, every value is compared to find out which one is the lowest.

The second time the algorithm runs through the array, every value **except the first value** is compared to find out which is the lowest.

And in this way the unsorted part of the array becomes shorter and shorter until the sorting is done. So on average,  $\frac{n}{2}$  elements are considered when the algorithm goes through the array finding the lowest value and moving it to the front of the array.

In addition to all the compares needed, the number of swaps needed is  $n$ .

We can start calculating the number of operations for the Selection Sort algorithm:

$$= \frac{n^4}{2} + \frac{n}{2}$$

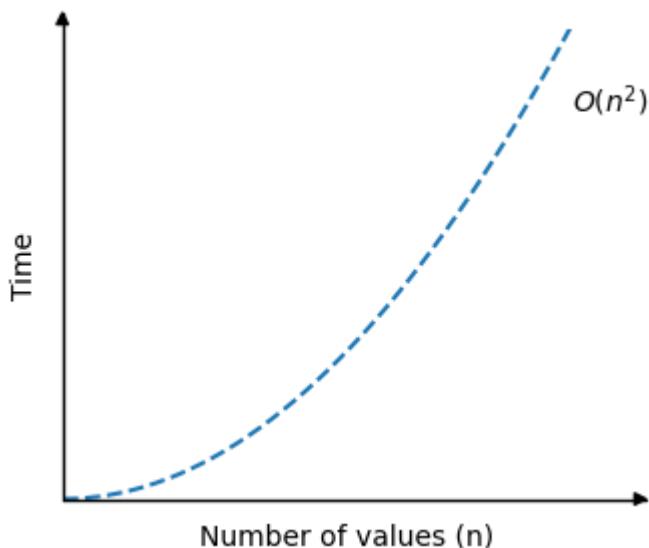
When looking at the run time for algorithms, we look at very large data sets, meaning  $n$  is a very big number. And for a very big number  $n$ , the term  $\frac{n^2}{2}$  becomes so much bigger than the term  $\frac{n}{2}$  that we can approximate by simply removing that second term  $\frac{n}{2}$ .

$$\text{Operations} = \frac{n^2}{2} + \frac{n}{2} \approx \frac{n^2}{2} = \frac{1}{2} \cdot n^2$$

Using Big O notation to describe the time complexity for the Selection Sort algorithm, we get:

$$O\left(\frac{1}{2} \cdot n^2\right) = \underline{\underline{O(n^2)}}$$

And time complexity for the Selection Sort algorithm can be displayed in a graph like this:



As you can see, the run time is the same as for Bubble Sort: The run time increases really fast when the size of the array is increased.

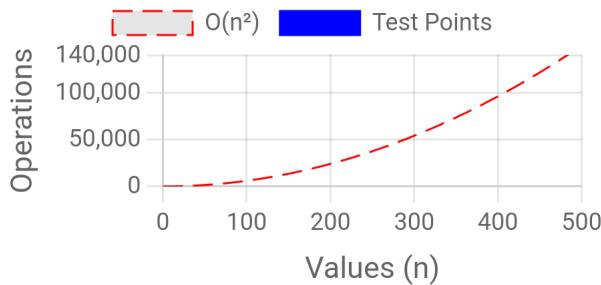
## Selection Sort Simulation

Run the simulation for different number of values in an array, and see how the number of operations Selection Sort needs on an array of  $n$  elements is  $O(n^2)$ :

Set values:  300

Best Case

Operations: 0

 10 RandomRunClear

The most significant difference from Bubble sort that we can notice in this simulation is that best and worst case is actually almost the same for Selection Sort ( $O(n^2)$ ), but for Bubble Sort the best case runtime is only  $O(n)$ .

The difference in best and worst case for Selection Sort is mainly the number of swaps. In the best case scenario Selection Sort does not have to swap any of the values because the array is already sorted. And in the worst case scenario, where the array already sorted, but in the wrong order, so Selection Sort must do as many swaps as there are values in the array.

[« Previous](#)[Next »](#)

W3schools Pathfinder  
Track your progress - it's free!

[Sign Up](#) [Log in](#)

ADVERTISEMENT

# DSA Insertion Sort Time Complexity

[« Previous](#)[Next »](#)

See [this page](#) for a general explanation of what time complexity is.

## Insertion Sort Time Complexity

The worst case scenario for Insertion Sort is if the array is already sorted, but with the highest values first. That is because in such a scenario, every new value must "move through" the whole sorted part of the array.

These are the operations that are done by the Insertion Sort algorithm for the first elements:

- The 1st value is already in the correct position.
- The 2nd value must be compared and moved past the 1st value.
- The 3rd value must be compared and moved past two values.
- The 3rd value must be compared and moved past three values.
- And so on..

If we continue this pattern, we get the total number of operations for  $n$  values:

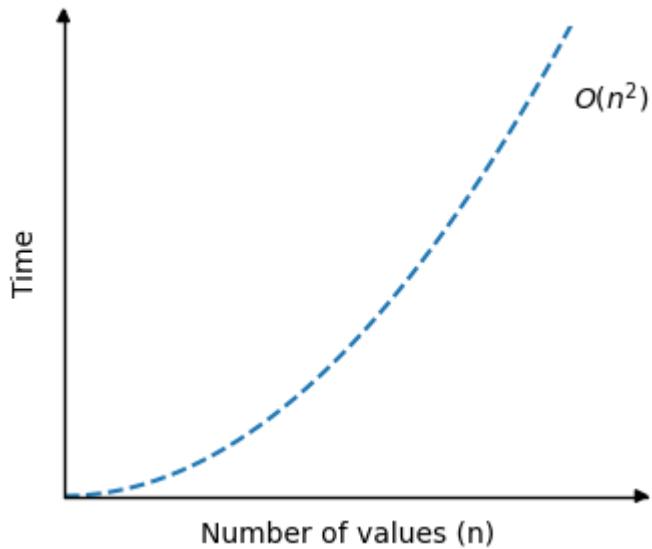
$$1 + 2 + 3 + \dots + (n - 1)$$

This is a well known series in mathematics that can be written like this:

$$\frac{n(n - 1)}{2} = \frac{n^2}{2} - \frac{n}{2}$$

For very large  $n$ , the  $\frac{n^2}{2}$  term dominates, so we can simplify by removing the second term  $\frac{n}{2}$ .

The time complexity can be displayed like this:



As you can see, the time used by Insertion Sort increases fast when the number of values is  $n$  increased.

## Insertion Sort Simulation

Use the simulation below to see how the theoretical time complexity  $O(n^2)$  (red line) compares with the number of operations of actual Insertion Sorts.

Set values:  300

Random

Worst Case

Best Case

10 Random

Operations: 0

Run Clear



For Insertion Sort, there is a big difference between best, average and worst case scenarios. You can see that by running the different simulations above.

The red line above represents the theoretical upper bound time complexity  $O(n^2)$ , and the actual function in this case is  $1.07 \cdot n^2$ .

Remember that a function  $f(n)$  is said to be  $O(g(n))$  if we have a positive constant  $C$  so that  $C \cdot g(n) > f(n)$ .

In this case  $f(n)$  is the number of operations used by Insertion Sort,  $g(n) = n^2$  and  $C = 1.07$ .

[« Previous](#)[Next »](#)

**W3schools Pathfinder**

Track your progress - it's free!

[Sign Up](#) [Log in](#)

ADVERTISEMENT

# DSA Time Complexity for Specific Algorithms

[« Previous](#)[Next »](#)

See [this page](#) for a general explanation of what time complexity is.

## Quicksort Time Complexity

The Quicksort algorithm chooses a value as the 'pivot' element, and moves the other values so that higher values are on the right of the pivot element, and lower values are on the left of the pivot element.

The Quicksort algorithm then continues to sort the sub-arrays on the left and right side of the pivot element recursively until the array is sorted.

## Worst Case

To find the time complexity for Quicksort, we can start by looking at the worst case scenario.

The worst case scenario for Quicksort is if the array is already sorted. In such a scenario, there is only one sub-array after each recursive call, and new sub-arrays are only one element shorter than the previous array.

This means that Quicksort must call itself recursively  $n$  times, and each time it must do  $\frac{n}{2}$  comparisons on average.

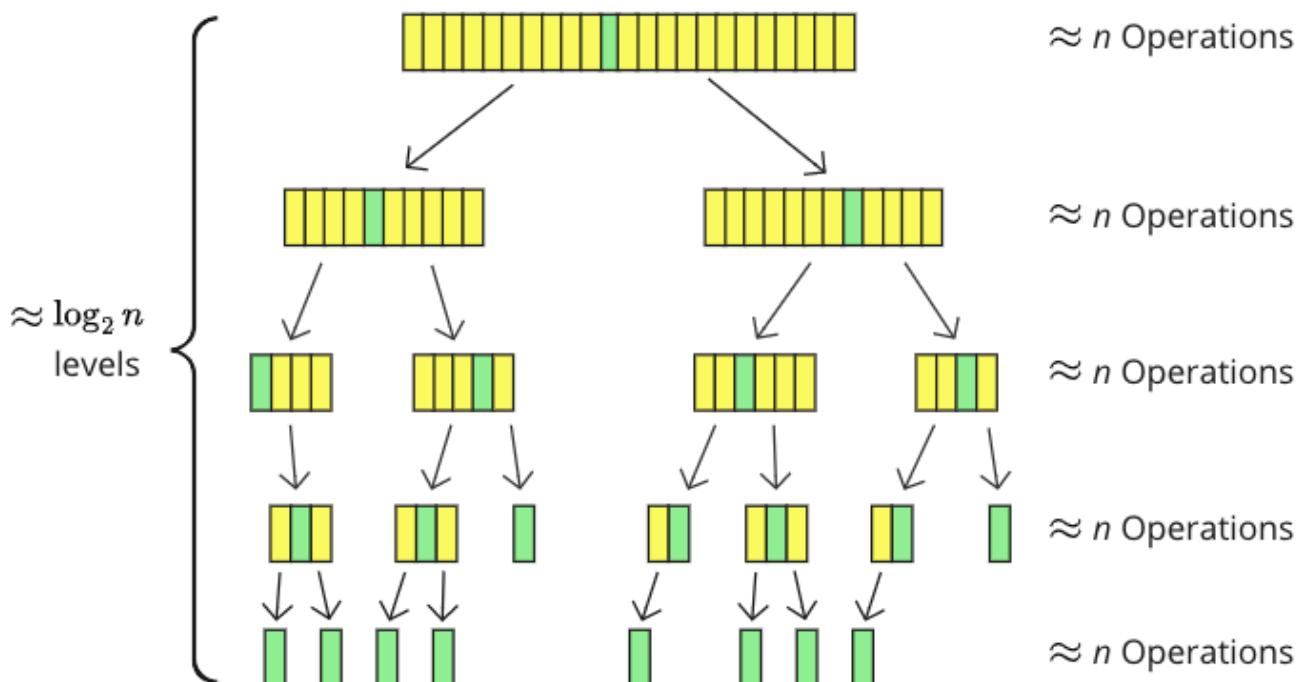
Worst case time complexity is:

# Average Case

On average, Quicksort is actually much faster.

Quicksort is fast on average because the array is split approximately in half each time Quicksort runs recursively, and so the size of the sub-arrays decrease really fast, which means that not so many recursive calls are needed, and Quicksort can finish sooner than in the worst case scenario.

The image below shows how an array of 23 values is split into sub-arrays when sorted with Quicksort.

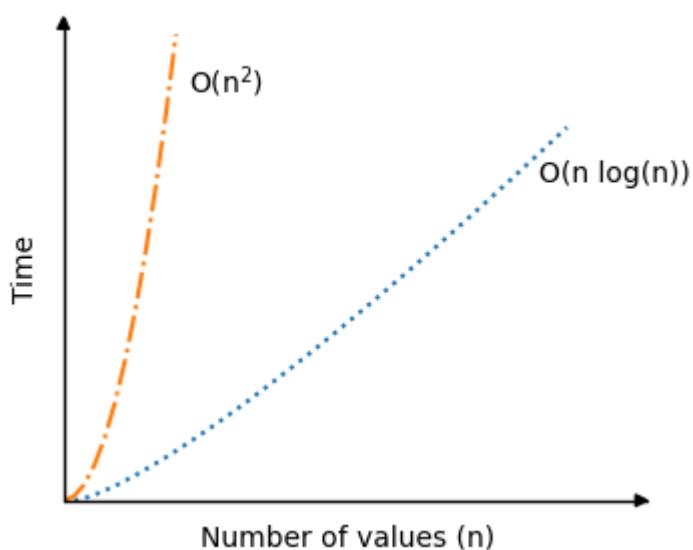


The pivot element (green) is moved into the middle, and the array is split into sub-arrays (yellow). There are 5 recursion levels with smaller and smaller sub-arrays, where about  $n$  values are touched somehow on each level: compared, or moved, or both.

$\log_2$  tells us how many times a number can be split in 2, so  $\log_2$  is a good estimate for how many levels of recursions there are.  $\log_2(23) \approx 4.5$  which is a good enough approximation of the number of recursion levels in the specific example above.

In reality, the sub-arrays are not split exactly in half each time, and there are not exactly  $n$  values compared or moved on each level, but we can say that this is the average case to find the time complexity:

$$\underline{O(n \cdot \log_2 n)}$$



The recursion part of the Quicksort algorithm is actually a reason why the average sorting scenario is so fast, because for good picks of the pivot element, the array will be split in half somewhat evenly each time the algorithm calls itself. So the number of recursive calls do not double, even if the number of values  $n$  double.

## Quicksort Simulation

Use the simulation below to see how the theoretical time complexity  $O(n^2)$  (red line) compares with the number of operations of actual Quicksort runs.

Set values:  300

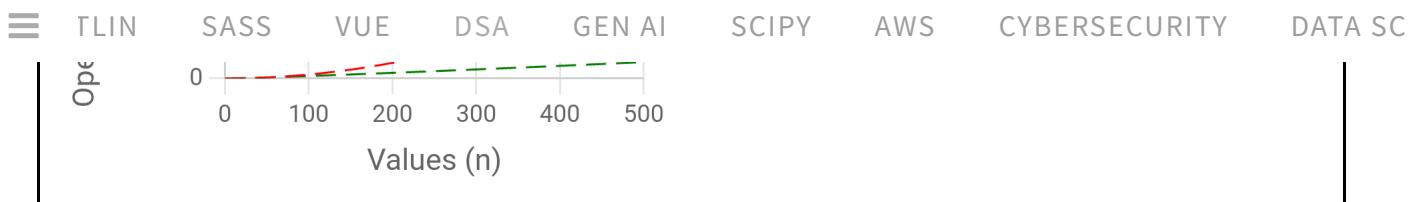
Random

Descending

Ascending

10 Random

Operations: 0



The red line above represents the theoretical upper bound time complexity  $O(n^2)$  for the worst case scenario, and the green line represents the average case scenario time complexity with random values  $O(n \log_2 n)$ .

For Quicksort, there is a big difference between average random case scenarios and scenarios where arrays are already sorted. You can see that by running the different simulations above.

The reason why the already ascending sorted array needs so many operations is that it requires the most swapping of elements, because of the way it is implemented. In this case, the last element is chosen as the pivot element, and the last element is also the highest number. So all other values in every sub-array is swapped around to land on the left side of the pivot element (where they are positioned already).

[« Previous](#)[Next »](#)

**W3schools Pathfinder**  
Track your progress - it's free!

[Sign Up](#) [Log in](#)

ADVERTISEMENT

# DSA Counting Sort Time Complexity

[« Previous](#)[Next »](#)

See [this page](#) for a general explanation of what time complexity is.

## Counting Sort Time Complexity

Counting Sort works by first counting the occurrence of different values, and then uses that to recreate the array in a sorted order.

As a rule of thumb, the Counting Sort algorithm runs fast when the range of possible values  $k$  is smaller than the number of values  $n$ .

To represent the time complexity with Big O notation we need to first count the number of operations the algorithm does:

- Finding the maximum value: Every value must be evaluated once to find out if it is the maximum value, so  $n$  operations are needed.
- Initializing the counting array: With  $k$  as the maximum value in the array, we need  $k + 1$  elements in the counting array to include 0. Every element in the counting array must be initialized, so  $k + 1$  operations are needed.
- Every value we want to sort is counted once, then removed, so 2 operations per count,  $2 \cdot n$  operations in total.
- Building the sorted array: Create  $n$  elements in the sorted array:  $n$  operations.

In total we get:

Based on what we have seen about time complexity earlier, we can create a simplified expression using Big O notation to represent time complexity:

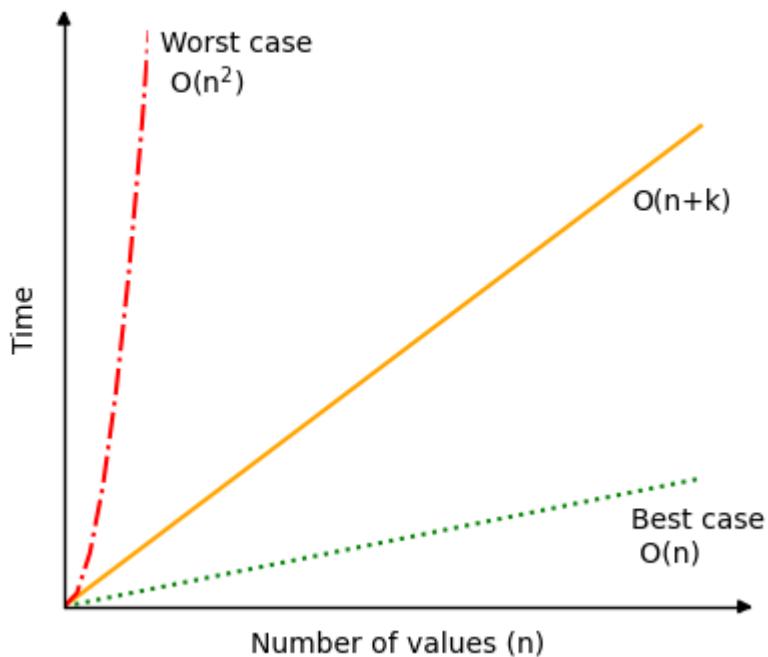
$$\begin{aligned}
 O(4 \cdot n + k) &= O(4 \cdot n) + O(k) \\
 &= O(n) + O(k) \\
 &= \underline{\underline{O(n+k)}}
 \end{aligned}$$

It has already been mentioned that Counting Sort is effective when the range of different values  $k$  is relatively small compared to the total number of values to be sorted  $n$ . We can now see this directly from the Big O expression  $O(n+k)$ .

Just imagine for example that the range of different numbers  $k$  is 10 times as large as the number of the values to be sorted. In such a case we can see that the algorithm uses most of its time on handling the range of different numbers  $k$ , although the actual number of values to be sorted  $n$  is small in comparison.

It is not straight forward to show the time complexity for Counting Sort in a graph, or to have a simulation for the time complexity as we have had for the previous algorithms, because the time complexity is so greatly affected by the range of values  $k$ .

Below is a plot that shows how much the time complexity for Counting Sort can vary, followed by an explanation for the best and worst case scenarios.



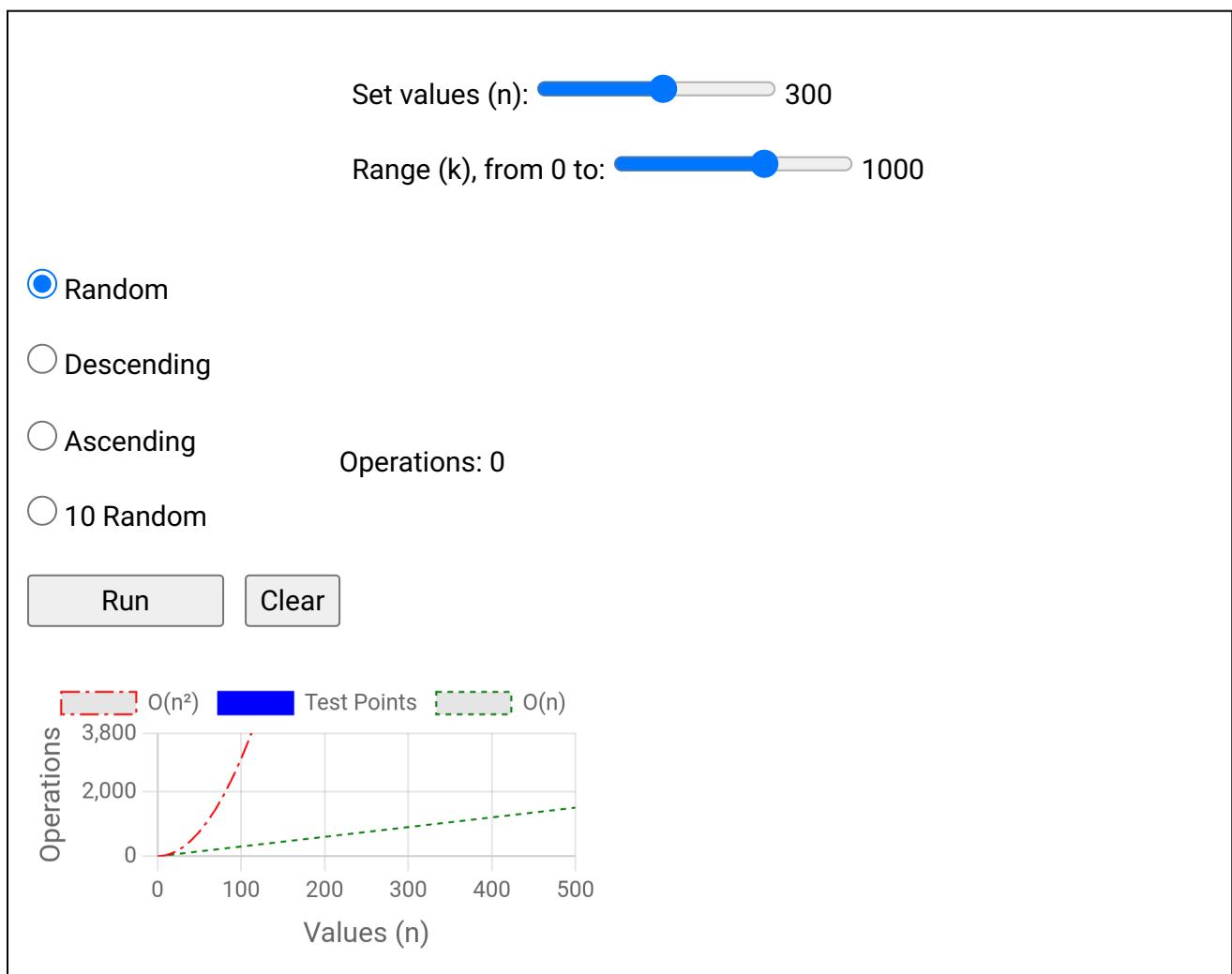
The **best case** scenario for Counting Sort would be to have the range  $k$  just a fraction of  $n$ , let's say  $k(n) = 0.1 \cdot n$ . As an example of this, for 100 values, the range would be from 0 to 10, or for

just 10 values the the range is between 0 and 100, or similarly, for an input of 1000 values, the range is between 0 and 1000000. In such a scenario, the growth of  $k$  is quadratic with respect to  $n$ , like this:  $k(n) = n^2$ , and we get time complexity  $O(n + k) = O(n + n^2)$  which is simplified to  $O(n^2)$ . A case that is even worse than this could also be constructed, but this case is chosen because it is relatively easy to understand, and perhaps not that unrealistic either.

As you can see, it is important to consider the range of values compared to the number of values to be sorted before choosing Counting Sort as your algorithm. Also, as mentioned at the top of the page, keep in mind that Counting sort only works for non negative integer values.

## Counting Sort Simulation

Run different simulations of Counting Sort to see how the number of operations falls between the worst case scenario  $O(n^2)$  (red line) and best case scenario  $O(n)$  (green line).



As mentioned previously: if the numbers to be sorted varies a lot in value (large  $k$ ), and there are few numbers to sort (small  $n$ ), the Counting Sort algorithm is not effective.

# DSA Merge Sort Time Complexity

[« Previous](#)[Next »](#)

See [this page](#) for a general explanation of what time complexity is.

## Merge Sort Time Complexity

The [Merge Sort algorithm](#) breaks the array down into smaller and smaller pieces.

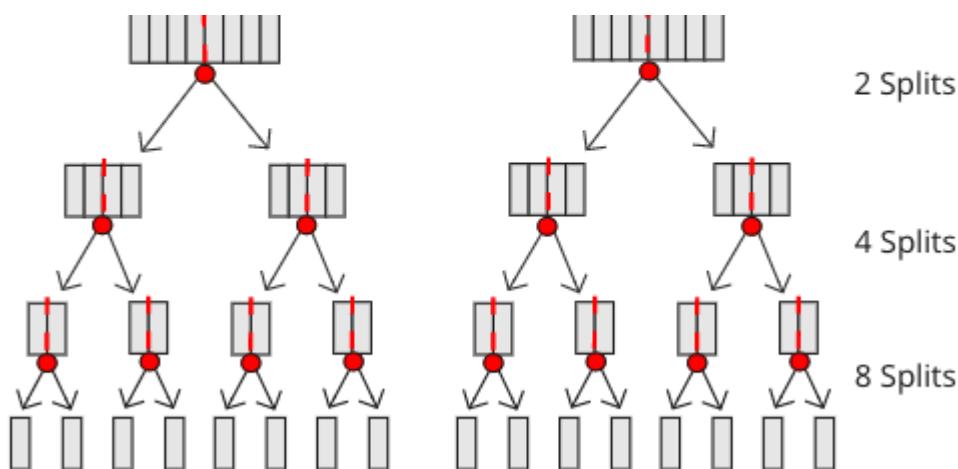
The array becomes sorted when the sub-arrays are merged back together so that the lowest values come first.

The array that needs to be sorted has  $n$  values, and we can find the time complexity by start looking at the number of operations needed by the algorithm.

The main operations Merge Sort does is to split, and then merge by comparing elements.

To split an array from start until the sub-arrays only consists of one value, Merge Sort does a total of  $n - 1$  splits. Just imaging an array with 16 values. It is split one time into sub-arrays of length 8, split again and again, and the size of the sub-arrays reduces to 4, 2 and finally 1. The number of splits for an array of 16 elements is  $1 + 2 + 4 + 8 = 15$ .

The image below shows that 15 splits are needed for an array of 16 numbers.



The number of merges is actually also  $n - 1$ , the same as the number of splits, because every split needs a merge to build the array back together. And for each merge there is a comparison between values in the sub-arrays so that the merged result is sorted.

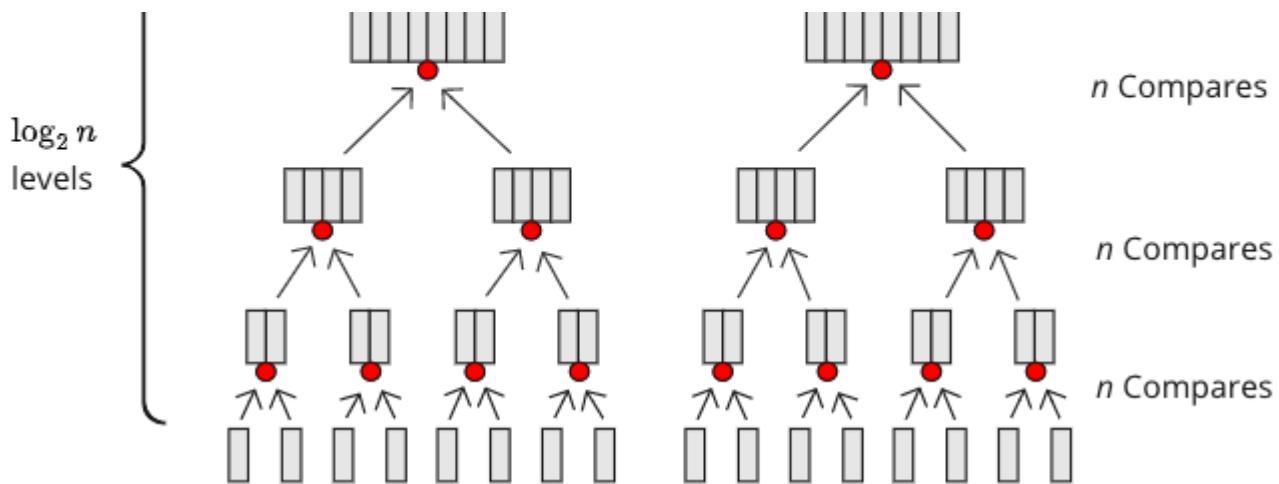
During merging of two sub-arrays, the worst case scenario that generates the most comparisons, is if the sub-arrays are equally big. Just consider merging [1,4,6,9] and [2,3,7,8]. In this case the following comparisons are needed:

1. Comparing 1 and 2, result: [1]
2. Comparing 4 and 2, result: [1,2]
3. Comparing 4 and 3, result: [1,2,3]
4. Comparing 4 and 7, result: [1,2,3,4]
5. Comparing 6 and 7, result: [1,2,3,4,6]
6. Comparing 9 and 7, result: [1,2,3,4,6,7]
7. Comparing 9 and 8, result: [1,2,3,4,6,7,8]

At the end of the merge, only the value 9 is left in one array, the other array is empty, so no comparison is needed to put the last value in, and the resulting merged array is [1,2,3,4,6,7,8,9]. We see that we need 7 comparisons to merge 8 values (4 values in each of the initial sub-arrays). In general, in a worst case scenario,  $n - 1$  comparisons are needed to get a merged array of  $n$  values.

For simplicity, let's say that we need  $n$  comparisons instead of  $n - 1$  when merging  $n$  values. This is an ok assumption when  $n$  is large and we want to calculate an upper bound using Big O notation.

So, at each level merging happens,  $n$  comparisons are needed, but how many levels are there? Well, for  $n = 16$  we have  $n = 16 = 2^4$ , so 4 levels of merging. For  $n = 32 = 2^5$  there are 5 levels of merging, and at each level,  $n$  comparisons are needed. For  $n = 1024 = 2^{10}$  10 levels of merging is needed. To find out what power of 2 gives us 1024, we use a base-2 logarithm. The answer is 10. Mathematically it looks like this:  $\log_2 1024 = 10$ .



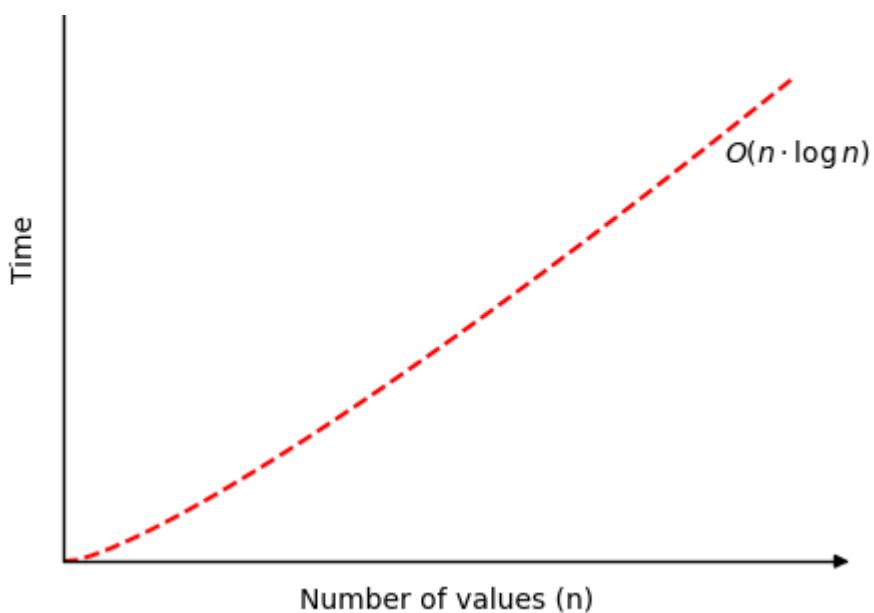
As we can see from the figure above,  $n$  comparisons are needed on each level, and there are  $\log_2 n$  levels, so there are  $n \cdot \log_2 n$  comparison operations in total.

Time complexity can be calculated based on the number of split operations and the number of merge operations:

$$O((n - 1) + n \cdot \log_2 n) = O(n \cdot \log_2 n)$$

The number of splitting operations ( $n - 1$ ) can be removed from the Big O calculation above because  $n \cdot \log_2 n$  will dominate for large  $n$ , and because of how we calculate time complexity for algorithms.

The figure below shows how the time increases when running Merge Sort on an array with  $n$  values.



The difference between best and worst case scenarios for Merge Sort is not as big as for many other sorting algorithms.

## Merge Sort Simulation

Run the simulation for different number of values in an array, and see how the number of operations Merge Sort needs on an array of  $n$  elements is  $O(n \log n)$ :

Set values:  300

Random

Descending

Ascending

10 Random

Operations: 0

Run Clear



If we hold the number of values  $n$  fixed, the number of operations needed for the "Random", "Descending" and "Ascending" is almost the same.

Merge Sort performs almost the same every time because the array is split, and merged using comparison, both if the array is already sorted or not.

[« Previous](#)[Next »](#)

**W3schools Pathfinder**  
Track your progress - it's free!

[Sign Up](#)    [Log in](#)

ADVERTISEMENT

Get Lifelong Access to  
**All Current & Future Courses**

Our certifications are recognised worldwide.

Original price: \$1,465  
**Discount price: \$695**

**Start Today!**



**COLOR PICKER**

# DSA Radix Sort Time Complexity

[« Previous](#)[Next »](#)

See [this page](#) for a general explanation of what time complexity is.

## Radix Sort Time Complexity

The [Radix Sort](#) algorithm sorts non negative integers, one digit at a time.

There are  $n$  values that need to be sorted, and  $k$  is the number of digits in the highest value.

When Radix Sort runs, every value is moved to the radix array, and then every value is moved back into the initial array. So  $n$  values are moved into the radix array, and  $n$  values are moved back. This gives us  $n + n = 2 \cdot n$  operations.

And the moving of values as described above needs to be done for every digit. This gives us a total of  $2 \cdot n \cdot k$  operations.

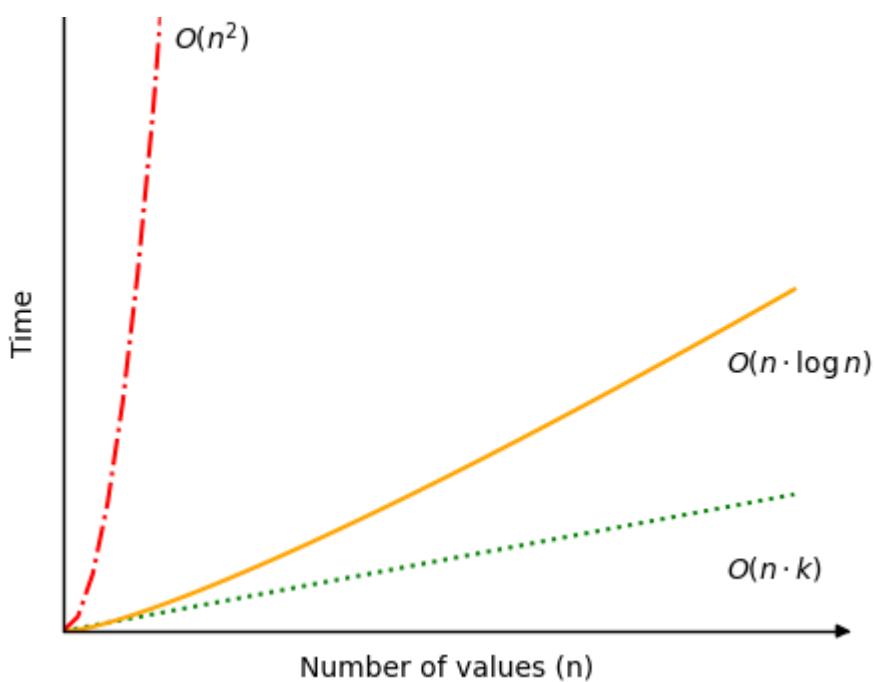
This gives us time complexity for Radix Sort:

$$O(2 \cdot n \cdot k) = \underline{\underline{O(n \cdot k)}}$$

Radix Sort is perhaps the fastest sorting algorithms there is, as long as the number of digits  $k$  is kept relatively small compared to the number of values  $n$ .

We can imagine a scenario where the number of digits  $k$  is the same as the number of values  $n$ , in such a case we get time complexity  $O(n \cdot k) = O(n^2)$  which is quite slow, and has the same time complexity as for example Bubble Sort.

We can also image a scenario where the number of digits  $k$  grow as the number of values  $n$  grow, so that  $k(n) = \log n$ . In such a scenario we get time complexity  $O(n \cdot k) = O(n \cdot \log n)$ , which is the same as for example Quicksort.



## Radix Sort Simulation

Run different simulations of Radix Sort to see how the number of operations falls between the worst case scenario  $O(n^2)$  (red line) and best case scenario  $O(n)$  (green line).

Set values (n):  300

Digits (k):  4

Random

Descending

Ascending

10 Random

Operations: 0



The bars representing the different values are scaled to fit the window, so that it looks ok. This means that values with 7 digits look like they are just 5 times bigger than values with 2 digits, but in reality, values with 7 digits are actually 5000 times bigger than values with 2 digits!

If we hold  $n$  and  $k$  fixed, the "Random", "Descending" and "Ascending" alternatives in the simulation above results in the same number of operations. This is because the same thing happens in all three cases.

[« Previous](#)[Next »](#)

**W3schools Pathfinder**  
Track your progress - it's free!

[Sign Up](#) [Log in](#)

ADVERTISEMENT

Get Lifelong Access to  
**All Current & Future Courses**

Our certifications are recognised worldwide.

Original price: \$1,465  
**Discount price: \$695**

[Start Today!](#)



# DSA Linear Search Time Complexity

[« Previous](#)[Next »](#)

See [this page](#) for a general explanation of what time complexity is.

## Linear Search Time Complexity

For a general explanation of what time complexity is, visit [this page](#).

For a more thorough and detailed explanation of Insertion Sort time complexity, visit [this page](#).

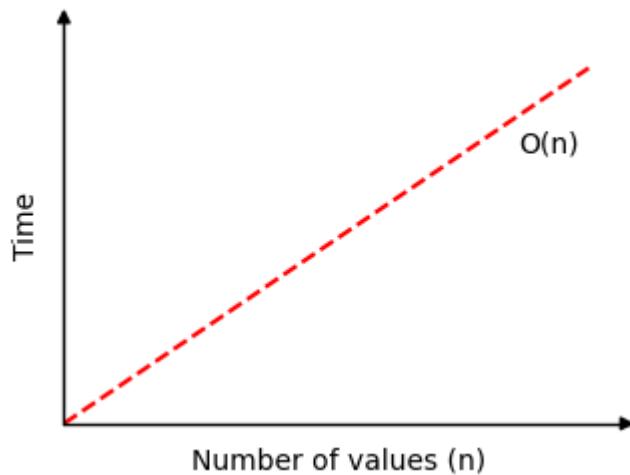
Linear Search compares each value with the value it is looking for. If the value is found, the index is returned, and if it is not found -1 is returned.

To find the time complexity for Linear Search, let's see if we can find out how many compare operations are needed to find a value in an array with  $n$  values.

**Best Case Scenario** is if the value we are looking for is the first value in the array. In such a case only one compare is needed and the time complexity is  $O(1)$ .

**Worst Case Scenario** is if the whole array is looked through without finding the target value. In such a case all values in the array are compared with the target value, and the time complexity is  $O(n)$ .

**Average Case Scenario** is not so easy to pinpoint. What is the possibility to finding the target value? That depends on the values in the array right? But if we assume that exactly one of the values in the array is equal to the target value, and that the position of that value can be anywhere, the average time needed for Linear Search is half of the time needed in the worst case scenario.



## Linear Search Simulation

Run the simulation for different number of values in an array, and see how many compares are needed for Linear Search to find a value in an array of  $n$  values:

Set values:  300

Random  
 Descending  
 Ascending  
 10 Random

Operations: 0

Run Clear

Operations

Values (n)

Legend:   $O(n)$   Found  Not found

Values (n)	Operations
0	0
100	100
200	200
300	300
400	400
500	500

# DSA Selection Sort Time Complexity

[« Previous](#)[Next »](#)

See [this page](#) for a general explanation of what time complexity is.

## Binary Search Time Complexity

Binary Search finds the target value in an already sorted array by checking the center value. If the center value is not the target value, Linear Search selects the left or right sub-array and continues the search until the target value is found.

To find the time complexity for Binary Search, let's see how many compare operations are needed to find the target value in an array with  $n$  values.

The **best case scenario** is if the first middle value is the same as the target value. If this happens the target value is found straight away, with only one compare, so the time complexity is  $O(1)$  in this case.

The **worst case scenario** is if the search area must be cut in half over and over until the search area is just one value. When this happens, it does not affect the time complexity if the target value is found or not.

Let's consider array lengths that are powers of 2, like 2, 4, 8, 16, 32, 64 and so on.

How many times must 2 be cut in half until we are looking at just one value? It is just one time, right?

How about 8? We must cut an array of 8 values in half 3 times to arrive at just one value.

An array of 32 values must be cut in half 5 times.

$\log_2(n)$  times.

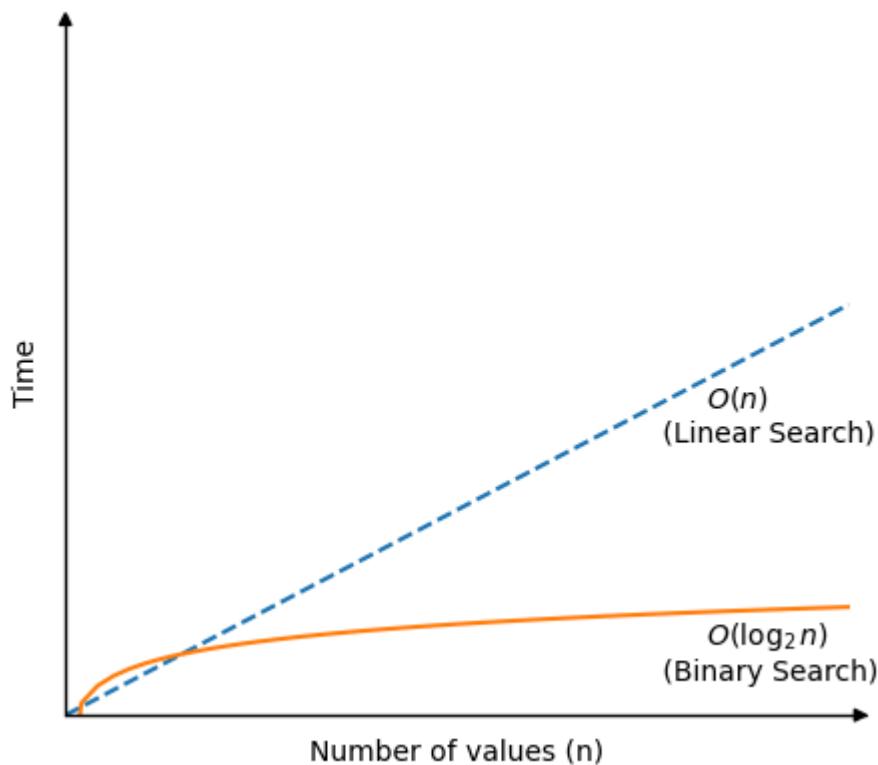
This means that time complexity for Binary Search is

$$\underline{\underline{O(\log_2 n)}}$$

The **average case scenario** is not so easy to pinpoint, but since we understand time complexity of an algorithm as the upper bound of the worst case scenario, using Big O notation, the average case scenario is not that interesting.

**Note:** Time complexity for Binary Search  $O(\log_2 n)$  is a lot faster than Linear Search  $O(n)$ , but it is important to remember that Binary Search requires a sorted array, and Linear Search does not.

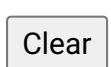
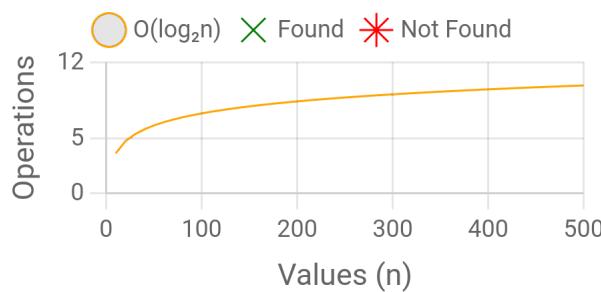
If we draw how much time Binary Search needs to find a value in an array of  $n$  values, compared to Linear Search, we get this graph:



## Binary Search Simulation

Set values:  300 Ascending 10 Ascending

Operations: 0

 Run Clear

As you can see when running simulations of Binary Search, the search requires very few compares, even if the array is big and the value we are looking for is not found.

 < PreviousNext >

W3schools Pathfinder

Track your progress - it's free!

[Sign Up](#)[Log in](#)

ADVERTISEMENT

[Tutorials ▾](#)[Exercises ▾](#)[Services ▾](#)[Sign Up](#)[Log in](#)[☰](#) [TLIN](#) [SASS](#) [VUE](#) [DSA](#) [GEN AI](#) [SCIPY](#) [AWS](#) [CYBERSECURITY](#) [DATA SC](#)

# The Euclidean Algorithm

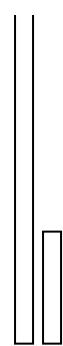
[« Previous](#)[Next »](#)

Named after the ancient Greek mathematician Euclid, the Euclidean algorithm is the oldest known non-trivial algorithm, described in Euclid's famous book "Elements" from 300 BCE.

## The Euclidean Algorithm

The Euclidean algorithm finds the greatest common divisor (gcd) of two numbers  $a$  and  $b$ .

The greatest common divisor is the largest number that divides both  $a$  and  $b$  without leaving a remainder.



$a = 50$

$b = 15$

Result:

▶ Calculations

The algorithm uses division with remainder. It takes the remainder from the previous step to set up the calculation in the next step.

#### How it works:

1. Start with the two initial numbers  $a$  and  $b$ .
2. Do a division with remainder:  $a = q_0 \cdot b + r_0$
3. Use the remainder ( $r_0$ ) and the divisor ( $b$ ) from the last calculation to set up the next calculation:  $b = q_1 \cdot r_0 + r_1$
4. Repeat steps 2 and 3 until the remainder is 0.
5. The second last remainder calculated is the greatest common divisor.

Continue reading to see how the Euclidean algorithm can be done by hand, with programming, and to understand how and why the algorithm actually works.

## Mathematical Terminology

a divisor of 6 because  $6/3 = 2$ , without leaving a remainder (the remainder is 0).

**Remainder:** The part you are left with after dividing a number with another number. Dividing 7 by 3 is 2, with a remainder of 1. (So 3 is not a divisor of 7.)

**Common divisor:** For numbers  $a$  and  $b$ , a common divisor is a number that can divide both  $a$  and  $b$  without leaving a remainder. The common divisors of 18 and 12 are 2, 3, and 6, because both 18 and 12 can be divided by 2, 3, and 6 without producing a remainder.

**Greatest common divisor:** The largest of the common divisors. The greatest common divisor of 18 and 12 is 6 because that is the largest of the common divisors 2, 3, and 6.

The greatest common divisor is used in the mathematical field of Number Theory, and in cryptography for encrypting messages.

**Note:** All numbers used by the Euclidean algorithm are integers.

## Manual Run Through

To understand how the Euclidean algorithm works, and to write the code for it, let's first run it manually to find the greatest common divisor of 120 and 25.

To do this we use division with remainder.

**Step 1:** We start with dividing 120 with 25:

$$120 = 4 \cdot 25 + 20$$

How many times can we fit 25 inside 120? It is 4 times, right?  $4 \cdot 25$  is 100. We get the remainder 20 by subtracting 100 from 120.

**Step 2:** We use the previous remainder 20 in the next step to divide 25:

$$25 = 1 \cdot 20 + 5$$

We can fit 20 inside 25 one time. We get the remainder 5 by subtracting 20 from 25.

**Step 3:** In the next calculation we divide 20 with the previous remainder 5:

$$20 = 4 \cdot 5 + 0$$

# Implementation of The Euclidean Algorithm

To find the greatest common divisor using division, we continue running the algorithm until the remainder calculated is 0.

This is the same as saying we continue to run the algorithm as long as  $b$  is not 0. That is why `b != 0` is the condition in the `while` loop below.

## Example

Finding the greatest common divisor of 120 and 25 using the Euclidean algorithm:

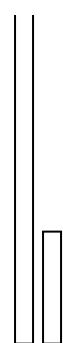
```
def gcd_division(a, b):
    while b != 0:
        remainder = a % b
        print(f"{a} = {a//b} * {b} + {remainder}")
        a = b
        b = remainder
    return a

a = 120
b = 25
print("The Euclidean algorithm using division:\n")
print(f"The GCD of {a} and {b} is: {gcd_division(a, b)}")
```

[Run Example »](#)

## The Original Euclidean Algorithm

Instead of using division like we did above, the original Euclidean algorithm as described in the book "Elements" over 2000 years ago uses subtraction.



$a = 50$

$b = 15$

Result:

▶ Calculations

### How the Euclidean algorithm with subtraction works:

1. Start with the two initial numbers  $a$  and  $b$ .
2. Find the difference  $a - b = c$ . The difference  $c$  shares the same greatest common divisor as  $a$  and  $b$ .
3. Take the two lowest numbers of  $a$ ,  $b$ , and  $c$ , and find the difference between them.
4. Repeat steps 2 and 3 until the difference is 0.
5. The second last difference calculated is the greatest common divisor.

Using subtraction instead of division is not as fast, but both the division method and the subtraction method uses the same mathematical principle:

*The greatest common divisor of numbers  $a$  and  $b$ , is also the greatest common divisor of the difference between  $a$  and  $b$ .*

This means that both  $a$  and  $b$  can be factorized like this:

$$a = k \cdot x$$

$$b = l \cdot x$$

After factorization, subtracting  $b$  from  $a$  gives us a very interesting result:

$$\begin{aligned} a - b &= k \cdot x - l \cdot x \\ &= (k - l) \cdot x \end{aligned}$$

We can see that the greatest common divisor ( $x$ ) of  $a$  and  $b$  is also the greatest common divisor of the difference between  $a$  and  $b$ !

This principle is why the Euclidean algorithm works, it is what makes it possible.

# Finding The Greatest Common Divisor Using Subtraction

Using the principle described above, that the difference between  $a$  and  $b$  also shares the same greatest common divisor, we can use subtraction to find the greatest common divisor, like Euclid's original algorithm does.

Let's find the greatest common divisor of 120 from 25 using subtraction.

$$120 - 25 = 95$$

According to the mathematical principle already described, the numbers 120, 25, and 95 all share the same greatest common divisor.

This means we can further reduce our problem by subtracting 25 from 95:

$$95 - 25 = 70$$

If we continue like this, always taking the two lowest numbers from the previous step and finding the difference between them, we get these calculations:

$$15 - 5 = 10$$

$$10 - 5 = \underline{5}$$

$$5 - 5 = 0$$

The Euclidean algorithm using subtraction is done when the difference is 0.

The greatest common divisor of 120 and 25 can be found in the previous step, it is 5.

Now that we can calculate the greatest common divisor using subtraction by hand, it is easier to implement it in a programming language.

## Implementation of The Euclidean Algorithm Using Subtraction

To find the greatest common divisor using subtraction, we continue running the algorithm until the difference between  $a$  and  $b$  is 0, like we have just seen.

This is the same as saying we continue running the algorithm as long as  $a$  and  $b$  are different values. That is why `a != b` is the condition in the `while` loop below.

### Example

Finding the greatest common divisor of 120 and 25 using the Euclidean algorithm with subtraction:

```
def gcd_subtraction(a, b):
    while a != b:
        if a > b:
            print(f"{a} - {b} = {a-b}")
            a = a - b
        else:
            print(f"{b} - {a} = {b-a}")
            b = b - a
    return a

a = 120
b = 25
```

# Comparing The Subtraction Method With The Division Method

To see how good the division method can be to find the greatest common divisor, and how the methods are similar, we will:

1. Use subtraction to find the greatest common divisor of 120 and 25.
  2. Use division with remainder to find the greatest common divisor of 120 and 25.
  3. Compare the subtraction and division methods.
- 

## 1. Using Subtraction

Finding the greatest common divisor of 120 and 25 using subtraction:

$$\begin{aligned}120 - 25 &= 95 \\95 - 25 &= 70 \\70 - 25 &= 45 \\45 - 25 &= 20 \\25 - 20 &= 5 \\20 - 5 &= 15 \\15 - 5 &= 10 \\10 - 5 &= \underline{5} \\5 - 5 &= 0\end{aligned}$$

Using subtraction, the algorithm is finished when the difference is 0.

In the second last calculation we see that the greatest common divisor of 120 and 25 is 5.

Notice how 25 and 5 must be subtracted many times, until finding the GCD.

---

## 2. Using Division

Finding the greatest common divisor of 120 and 25 using division with remainder looks like this:

Using division, the Euclidean algorithm is finished when the remainder is 0.

The previous remainder 5 is the greatest common divisor of 120 and 25.

### 3. Comparison

Take a look at the subtraction and division methods above.

To easier see that the division calculations are basically the same as the subtraction calculations, we can write the division with remainder calculations like this:

$$\begin{aligned}120 - 4 \cdot 25 &= 20 \\25 - 1 \cdot 20 &= \underline{5} \\20 - 4 \cdot 5 &= 0\end{aligned}$$

Using subtraction, 25 is subtracted from 120 a total of 4 times, while the division method does this in just one step.

Similarly, the subtraction method subtracts 5 a total of 4 times, while the division method does the same in just one calculation.

As you can see, the methods do the same thing, the division method just does many subtractions in the same calculation, so that it finds the greatest common divisor faster.

[« Previous](#)[Next »](#)

**W3schools Pathfinder**  
Track your progress - it's free!

[Sign Up](#)   [Log in](#)

ADVERTISEMENT

# Huffman Coding

[« Previous](#)[Next »](#)

## Huffman Coding

Huffman Coding is an algorithm used for lossless data compression.

Huffman Coding is also used as a component in many different compression algorithms. It is used as a component in lossless compressions such as zip, gzip, and png, and even as part of lossy compression algorithms like mp3 and jpeg.

Use the animation below to see how a text can be compressed using Huffman Coding.

Text:

Huffman code:	UTF-8:
	01101100 01101111
	01110011 01110011
	01101100 01100101
	01110011 01110011

64 bits

The animation shows how the letters in a text are normally stored using UTF-8, and how Huffman Coding makes it possible to store the same text with fewer bits.

## How it works:

1. Count how often each piece of data occurs.
2. Build a binary tree, starting with the nodes with the lowest count. The new parent node has the combined count of its child nodes.
3. The edge from a parent gets '0' for the left child, and '1' for the edge to the right child.
4. In the finished binary tree, follow the edges from the root node, adding '0' or '1' for each branch, to find the new Huffman code for each piece of data.
5. Create the Huffman code by converting the data, piece-by-piece, into a binary code using the binary tree.

Huffman Coding uses a variable length of bits to represent each piece of data, with a shorter bit representation for the pieces of data that occurs more often.

Furthermore, Huffman Coding ensures that no code is the prefix of another code, which makes the compressed data easy to decode.

**Data compression** is when the original data size is reduced, but the information is mostly, or fully, kept. Sound or music files are for example usually stored in a compressed format, roughly just 10% of the original data size, but with most of the information kept.

**Lossless** means that even after the data is compressed, all the information is still there. This means that for example a compressed text still has all the same letters and characters as the original.

**Lossy** is the other variant of data compression, where some of the original information is lost, or sacrificed, so that the data can be compressed even more. Music, images, and video is normally stored and streamed with lossy compression like mp3, jpeg, and mp4.

# Creating A Huffman Code Manually

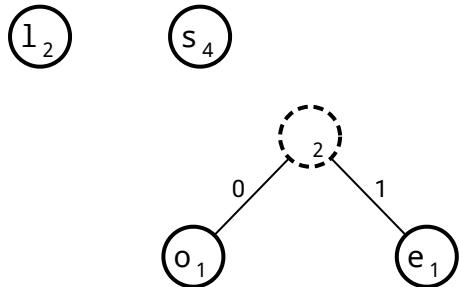
8 bits for normal latin letters, like we have in 'lossless'. Other letters or symbols such as '€' or '🦄' are stored using more bits.

To compress the text 'lossless' using Huffman Coding, we start by counting each letter.

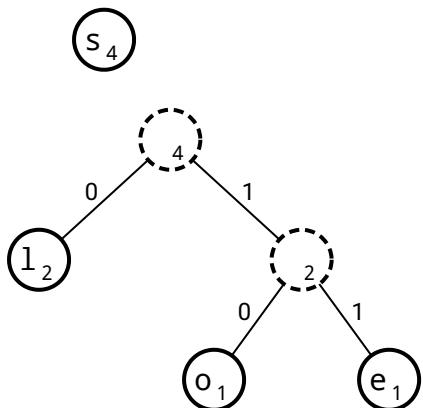


As you can see in the nodes above, 's' occurs 4 times, 'l' occurs 2 times, and 'o' and 'e' occurs just 1 time each.

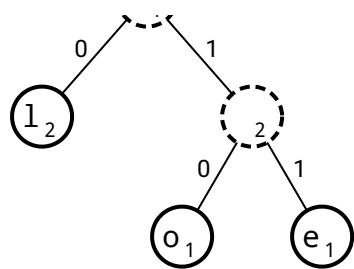
We start building the tree with the least occurring letters 'o' and 'e', and their parent node gets count '2', because the counts for letter 'o' and 'e' are summarized.



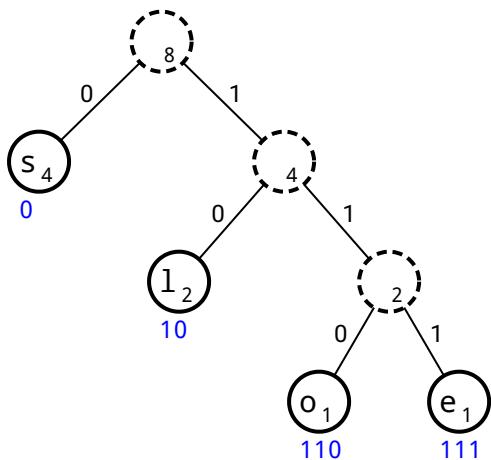
The next nodes that get a new parent node, are the nodes with the lowest count: 'l', and the parent node of 'o' and 'e'.



Now, the last node 's' must be added to the binary tree. Letter node 's' and the parent node with count '4' get a new parent node with count '8'.



Following the edges from the root node, we can now determine the Huffman code for each letter in the word 'lossless'.



The Huffman code for each letter can now be found under each letter node in the image above. A good thing about Huffman coding is that the most used data pieces get the shortest code, so just '0' is the code for the letter 's'.

As mentioned earlier, such normal latin letters are usually stored with UTF-8, which means they take up 8 bits each. So for example the letter 'o' is stored as '01101111' with UTF-8, but it is stored as '110' with our Huffman code for the word 'lossless'.

**Note:** With UTF-8, a letter has always the same binary code, but with Huffman code, the binary code for each letter (piece of data) changes with text (data set) we are compressing.

To summarize, we have now compressed the word 'lossless' from its UTF-8 code

01101100 01101111 01110011 01110011 01101100 01100101 01110011  
01110011

to just

10 110 0 0 10 111 0 0

Furthermore, the binary code is really `10110001011100`, without the spaces, and with variable bit lengths for each piece of data, so how can the computer understand where the binary code for each piece of data starts and ends?

## Decoding Huffman Code

Just like with code stored as UTF-8, which our computers can already decode to the correct letters, the computer needs to know which bits represent which piece of data in the Huffman code.

So along with a Huffman code, there must also be a conversion table with information about what the Huffman binary code is for each piece of data, so that it can be decoded.

So, for this Huffman code:

`100110110`

With this conversion table:

Letter	Huffman Code
a	0
b	10
n	11

Are you able to decode the Huffman code?

### How it works:

1. Start from the left in the Huffman code, and look up each bit sequence in the table.
2. Match each code to the corresponding letter.
3. Continue until the entire Huffman code is decoded.

We start with the first bit:

1 0 0 1 1 0 1 1 0

We can see from the table that 10 is 'b', so now we have the first letter. We check the next bit:

1 0 0 1 1 0 1 1 0

We find that 0 is 'a', so now we have the two first letters 'ba' stored in the Huffman code.

We continue looking up Huffman codes in the table:

1 0 0 1 1 0 1 1 0

Code 11 is 'n'.

1 0 0 1 1 0 1 1 0

Code 0 is 'a'.

1 0 0 1 1 0 1 1 0

Code 11 is 'n'.

1 0 0 1 1 0 1 1 0

Code 0 is 'a'.

The Huffman code is now decoded, and the word is 'banana'!

## Huffman Code Prefixes

An interesting and very useful part of the Huffman coding algorithm is that it ensures that there is no code that is the prefix of another code.

Just image if the conversion table we just used, looked like this:

Letter	Huffman Code
a	1

If this was the case, we would get confused right from the start of the decoding, right?

1 0 0 1 1 0 1 1 0

Because how would we know if the first bit `1` represents the letter 'a' or if it is the first bit for the letter 'b' or 'c'?

This property, that no code is the prefix of another code, makes it possible to decode. And it is especially important in Huffman Coding because of the variable bit lengths.

## Huffman Coding Implementation

The correct word for creating Huffman code based on data or text is "encoding", and the opposite would be "decoding", when the original data or text is recreated based on the code.

The code example below takes a word, or any text really, and compress it using Huffman Coding.

## Example

Huffman Coding.

```
class Node:
    def __init__(self, char=None, freq=0):
        self.char = char
        self.freq = freq
        self.left = None
        self.right = None

nodes = []

def calculate_frequencies(word):
    frequencies = {}
    for char in word:
        if char not in frequencies:
            freq = word.count(char)
            frequencies[char] = freq
            nodes.append(Node(char, freq))
```

☰ TLIN SASS VUE DSA GEN AI SCIPY AWS CYBERSECURITY DATA SC

```
        left = nodes.pop(0)
        right = nodes.pop(0)

        merged = Node(freq=left.freq + right.freq)
        merged.left = left
        merged.right = right

        nodes.append(merged)

    return nodes[0]

def generate_huffman_codes(node, current_code, codes):
    if node is None:
        return

    if node.char is not None:
        codes[node.char] = current_code

    generate_huffman_codes(node.left, current_code + '0', codes)
    generate_huffman_codes(node.right, current_code + '1', codes)

def huffman_encoding(word):
    global nodes
    nodes = []
    calculate_frequencies(word)
    root = build_huffman_tree()
    codes = {}
    generate_huffman_codes(root, '', codes)
    return codes

word = "lossless"
codes = huffman_encoding(word)
encoded_word = ''.join(codes[char] for char in word)

print("Word:", word)
print("Huffman code:", encoded_word)
print("Conversion table:", codes)
```

[Run Example »](#)

recreate the original information.

The implementation below is basically the same as the previous code example, but with an additional function for decoding the Huffman code.

The `huffman_decoding` function takes the Huffman code, and the `codes` [Python dictionary](#) (a [hashmap](#)) with the characters and their corresponding binary codes. The Function then reverse the mapping, and checks the Huffman code bit-by-bit to recreate the original text.

## Example

Huffman Decoding.

```
class Node:
    def __init__(self, char=None, freq=0):
        self.char = char
        self.freq = freq
        self.left = None
        self.right = None

nodes = []

def calculate_frequencies(word):
    frequencies = {}
    for char in word:
        if char not in frequencies:
            freq = word.count(char)
            frequencies[char] = freq
            nodes.append(Node(char, freq))

def build_huffman_tree():
    while len(nodes) > 1:
        nodes.sort(key=lambda x: x.freq)
        left = nodes.pop(0)
        right = nodes.pop(0)

        merged = Node(freq=left.freq + right.freq)
        merged.left = left
        merged.right = right

        nodes.append(merged)
```

```
if node is None:
    return

if node.char is not None:
    codes[node.char] = current_code

generate_huffman_codes(node.left, current_code + '0', codes)
generate_huffman_codes(node.right, current_code + '1', codes)

def huffman_encoding(word):
    global nodes
    nodes = []
    calculate_frequencies(word)
    root = build_huffman_tree()
    codes = {}
    generate_huffman_codes(root, '', codes)
    return codes

    return ''.join(encoded_chars)

word = "lossless"
codes = huffman_encoding(word)
encoded_word = ''.join(codes[char] for char in word)
decoded_word = huffman_decoding(encoded_word, codes)

print("Initial word:", word)
print("Huffman code:", encoded_word)
print("Conversion table:", codes)
```

[Tutorials ▾](#)[Exercises ▾](#)[Services ▾](#)[Sign Up](#)[Log in](#)[☰](#) [TLIN](#)[SASS](#)[VUE](#)[DSA](#)[GEN AI](#)[SCIPY](#)[AWS](#)[CYBERSECURITY](#)[DATA SC](#)

You have now seen how a text can be compressed using Huffman coding, and how a Huffman code can be decoded to recreate the original text.

**Note:** Huffman Coding can be used for lossless compression of any kind of data, not just text. Huffman Coding is also used as a component in other compression algorithms like zip, and even in lossy compressions like jpeg and mp3.

[« Previous](#)[Next »](#)

**W3schools Pathfinder**

Track your progress - it's free!

[Sign Up](#)[Log in](#)

ADVERTISEMENT

Get Lifelong Access to  
**All Current & Future Courses**

Our certifications are recognised worldwide.

Original price: \$1,465  
**Discount price: \$695**

[Start Today!](#)



# DSA The Traveling Salesman Problem

[« Previous](#)[Next »](#)

## The Traveling Salesman Problem

The Traveling Salesman Problem states that you are a salesperson and you must visit a number of cities or towns.

### The Traveling Salesman Problem

Rules: Visit every city only once, then return back to the city you started in.

Goal: Find the shortest possible route.

Except for the Held-Karp algorithm (which is quite advanced and time consuming,  $O(2^n n^2)$ ), and will not be described here), there is no other way to find the shortest route than to check all possible routes.

This means that the time complexity for solving this problem is  $O(n!)$ , which means 720 routes needs to be checked for 6 cities, 40,320 routes must be checked for 8 cities, and if you have 10 cities to visit, more than 3.6 million routes must be checked!

**Note:** "!", or "factorial", is a mathematical operation used in combinatorics to find out how many possible ways something can be done. If there are 4 cities, each city is connected to every other

The Traveling Salesman Problem (TSP) is a problem that is interesting to study because it is very practical, but so time consuming to solve, that it becomes nearly impossible to find the shortest route, even in a graph with just 20-30 vertices.

If we had an effective algorithm for solving The Traveling Salesman Problem, the consequences would be very big in many sectors, like for example chip design, vehicle routing, telecommunications, and urban planning.

## Checking All Routes to Solve The Traveling Salesman Problem

To find the optimal solution to The Traveling Salesman Problem, we will check all possible routes, and every time we find a shorter route, we will store it, so that in the end we will have the shortest route.

**Good:** Finds the overall shortest route.

**Bad:** Requires an awful lot of calculation, especially for a large amount of cities, which means it is very time consuming.

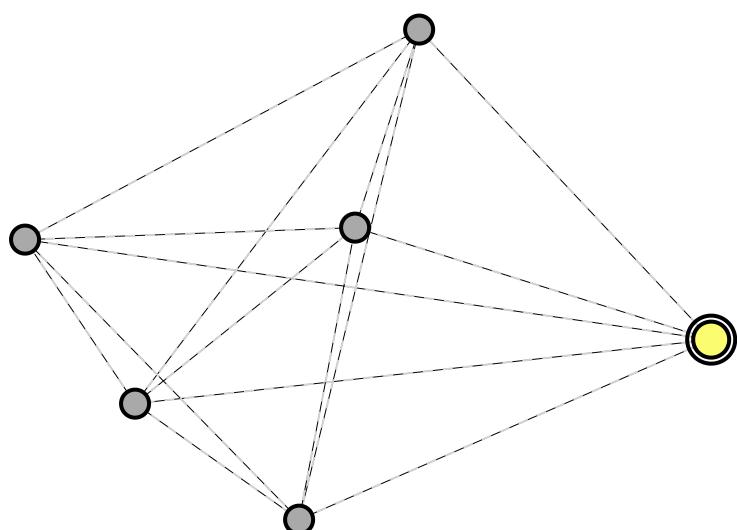
### How it works:

1. Check the length of every possible route, one route at a time.
2. Is the current route shorter than the shortest route found so far? If so, store the new shortest route.
3. After checking all routes, the stored route is the shortest one.

Such a way of finding the solution to a problem is called **brute force**.

Brute force is not really an algorithm, it just means finding the solution by checking all possibilities, usually because of a lack of a better way to do it.

Finding the shortest route in The Traveling Salesman Problem by checking all routes (brute force).



Progress: 0%

► Route distance:

n = 6 cities

$6! = 720$  possible routes

Show every route:  true

**Find Shortest Route**

**Reset**

The reason why the brute force approach of finding the shortest route (as shown above) is so time consuming is that we are checking all routes, and the number of possible routes increases really fast when the number of cities increases.

## Example

Finding the optimal solution to the Traveling Salesman Problem by checking all possible routes (brute force):

```
from itertools import permutations
```

☰ TLIN SASS VUE DSA GEN AI SCIPY AWS CYBERSECURITY DATA SC

```
total_distance += distances[route[-1]][route[0]]
return total_distance

def brute_force_tsp(distances):
    n = len(distances)
    cities = list(range(1, n))
    shortest_route = None
    min_distance = float('inf')

    for perm in permutations(cities):
        current_route = [0] + list(perm)
        current_distance = calculate_distance(current_route, distances)

        if current_distance < min_distance:
            min_distance = current_distance
            shortest_route = current_route

    shortest_route.append(0)
    return shortest_route, min_distance

distances = [
    [0, 2, 2, 5, 9, 3],
    [2, 0, 4, 6, 7, 8],
    [2, 4, 0, 8, 6, 3],
    [5, 6, 8, 0, 4, 9],
    [9, 7, 6, 4, 0, 10],
    [3, 8, 3, 9, 10, 0]
]

route, total_distance = brute_force_tsp(distances)
print("Route:", route)
print("Total distance:", total_distance)
```

[Run Example »](#)

# Using A Greedy Algorithm to Solve The Traveling Salesman Problem

**Good:** Finds a solution to the Traveling Salesman Problem much faster than by checking all routes.

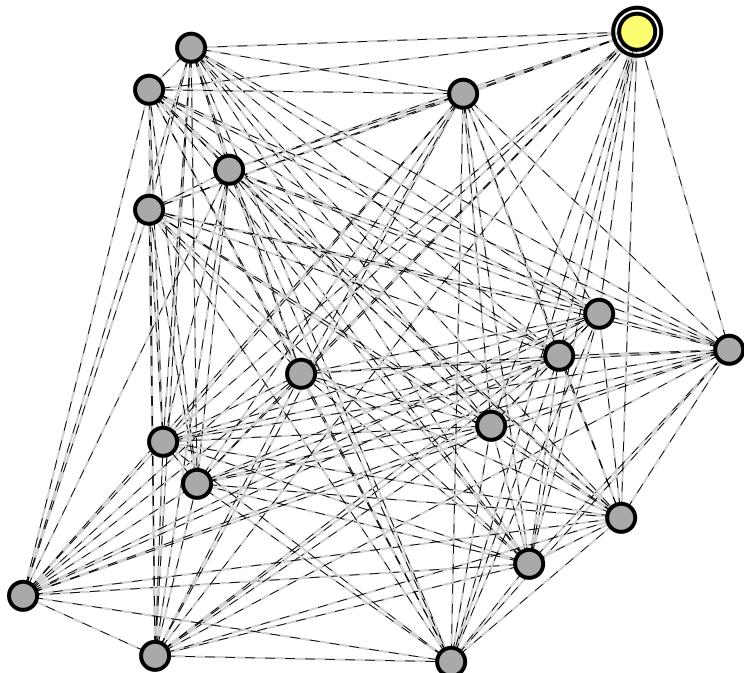
**Bad:** Does not find the overall shortest route, it just finds a route that is much shorter than an average random route.

### How it works:

1. Visit every city.
2. The next city to visit is always the nearest of the unvisited cities from the city you are currently in.
3. After visiting all cities, go back to the city you started in.

This way of finding an approximation to the shortest route in the Traveling Salesman Problem, by just going to the nearest unvisited city in each step, is called a **greedy algorithm**.

Finding an approximation to the shortest route in The Traveling Salesman Problem by always going to the nearest unvisited neighbor (greedy algorithm).



Greedy Algorithm

## Example

Finding a near-optimal solution to the Traveling Salesman Problem using the nearest-neighbor algorithm (greedy):

```
def nearest_neighbor_tsp(distances):
    n = len(distances)
    visited = [False] * n
    route = [0]
    visited[0] = True
    total_distance = 0

    for _ in range(1, n):
        last = route[-1]
        nearest = None
        min_dist = float('inf')
        for i in range(n):
            if not visited[i] and distances[last][i] < min_dist:
                min_dist = distances[last][i]
                nearest = i
        route.append(nearest)
        visited[nearest] = True
        total_distance += min_dist

    total_distance += distances[route[-1]][0]
    route.append(0)
    return route, total_distance

distances = [
    [0, 2, 2, 5, 9, 3],
    [2, 0, 4, 6, 7, 8],
    [2, 4, 0, 8, 6, 3],
    [5, 6, 8, 0, 4, 9],
    [9, 7, 6, 4, 0, 10],
    [3, 8, 3, 9, 10, 0]
]

route, total_distance = nearest_neighbor_tsp(distances)
```

# Other Algorithms That Find Near-Optimal Solutions to The Traveling Salesman Problem

In addition to using a greedy algorithm to solve the Traveling Salesman Problem, there are also other algorithms that can find approximations to the shortest route.

These algorithms are popular because they are much more effective than to actually check all possible solutions, but as with the greedy algorithm above, they do not find the overall shortest route.

Algorithms used to find a near-optimal solution to the Traveling Salesman Problem include:

- **2-opt Heuristic:** An algorithm that improves the solution step-by-step, in each step removing two edges and reconnecting the two paths in a different way to reduce the total path length.
- **Genetic Algorithm:** This is a type of algorithm inspired by the process of natural selection and use techniques such as selection, mutation, and crossover to evolve solutions to problems, including the TSP.
- **Simulated Annealing:** This method is inspired by the process of annealing in metallurgy. It involves heating and then slowly cooling a material to decrease defects. In the context of TSP, it's used to find a near-optimal solution by exploring the solution space in a way that allows for occasional moves to worse solutions, which helps to avoid getting stuck in local minima.
- **Ant Colony Optimization:** This algorithm is inspired by the behavior of ants in finding paths from the colony to food sources. It's a more complex probabilistic technique for solving computational problems which can be mapped to finding good paths through graphs.

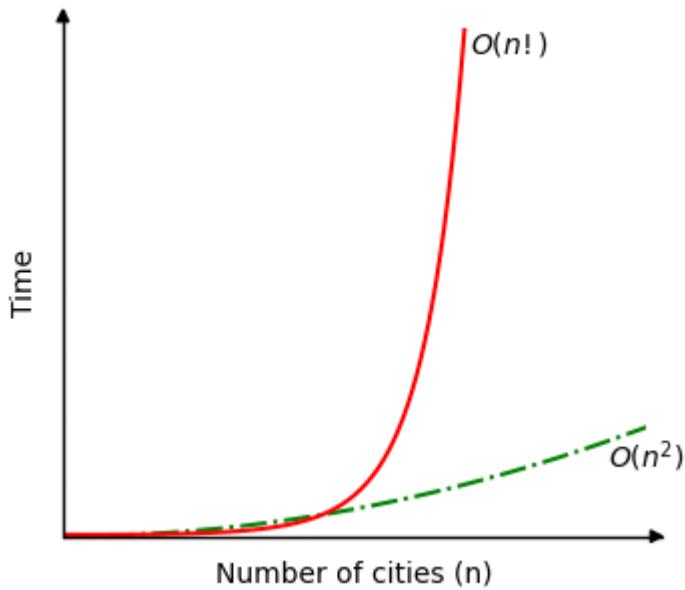
## Time Complexity for Solving The Traveling Salesman Problem

To get a near-optimal solution fast, we can use a greedy algorithm that just goes to the nearest unvisited city in each step, like in the second simulation on this page.

Solving The Traveling Salesman Problem in a greedy way like that, means that at each step, the distances from the current city to all other unvisited cities are compared, and that gives us a time

which is the same as  $4 \cdot 3 \cdot 2 \cdot 1 = 24$ . And for just 12 cities for example, there are  $12! = 12 \cdot 11 \cdot 10 \cdot \dots \cdot 2 \cdot 1 = 479,001,600$  possible routes!

See the time complexity for the greedy algorithm  $O(n^2)$ , versus the time complexity for finding the shortest route by comparing all routes  $O(n!)$ , in the image below.



But there are two things we can do to reduce the number of routes we need to check.

In the Traveling Salesman Problem, the route starts and ends in the same place, which makes a cycle. This means that the length of the shortest route will be the same no matter which city we start in. That is why we have chosen a fixed city to start in for the simulation above, and that reduces the number of possible routes from  $n!$  to  $(n - 1)!$ .

Also, because these routes go in cycles, a route has the same distance if we go in one direction or the other, so we actually just need to check the distance of half of the routes, because the other half will just be the same routes in the opposite direction, so the number of routes we need to check is actually  $\frac{(n-1)!}{2}$ .

But even if we can reduce the number of routes we need to check to  $\frac{(n-1)!}{2}$ , the time complexity is still  $O(n!)$ , because for very big  $n$ , reducing  $n$  by one and dividing by 2 does not make a significant change in how the time complexity grows when  $n$  is increased.

To better understand how time complexity works, go to [this page](#).

The edge weight in a graph in this context of The Traveling Salesman Problem tells us how hard it is to go from one point to another, and it is the total edge weight of a route we want to minimize.

So far on this page, the edge weight has been the distance in a straight line between two points. And that makes it much easier to explain the Traveling Salesman Problem, and to display it.

But in the real world there are many other things that affects the edge weight:

1. **Obstacles:** When moving from one place to another, we normally try to avoid obstacles like trees, rivers, houses for example. This means it is longer and takes more time to go from A to B, and the edge weight value needs to be increased to factor that in, because it is not a straight line anymore.
2. **Transportation Networks:** We usually follow a road or use public transport systems when traveling, and that also affects how hard it is to go (or send a package) from one place to another.
3. **Traffic Conditions:** Travel congestion also affects the travel time, so that should also be reflected in the edge weight value.
4. **Legal and Political Boundaries:** Crossing border for example, might make one route harder to choose than another, which means the shortest straight line route might be slower, or more costly.
5. **Economic Factors:** Using fuel, using the time of employees, maintaining vehicles, all these things cost money and should also be factored into the edge weights.

As you can see, just using the straight line distances as the edge weights, might be too simple compared to the real problem. And solving the Traveling Salesman Problem for such a simplified problem model would probably give us a solution that is not optimal in a practical sense.

It is not easy to visualize a Traveling Salesman Problem when the edge length is not just the straight line distance between two points anymore, but the computer handles that very well.

[◀ Previous](#)[Next ▶](#)

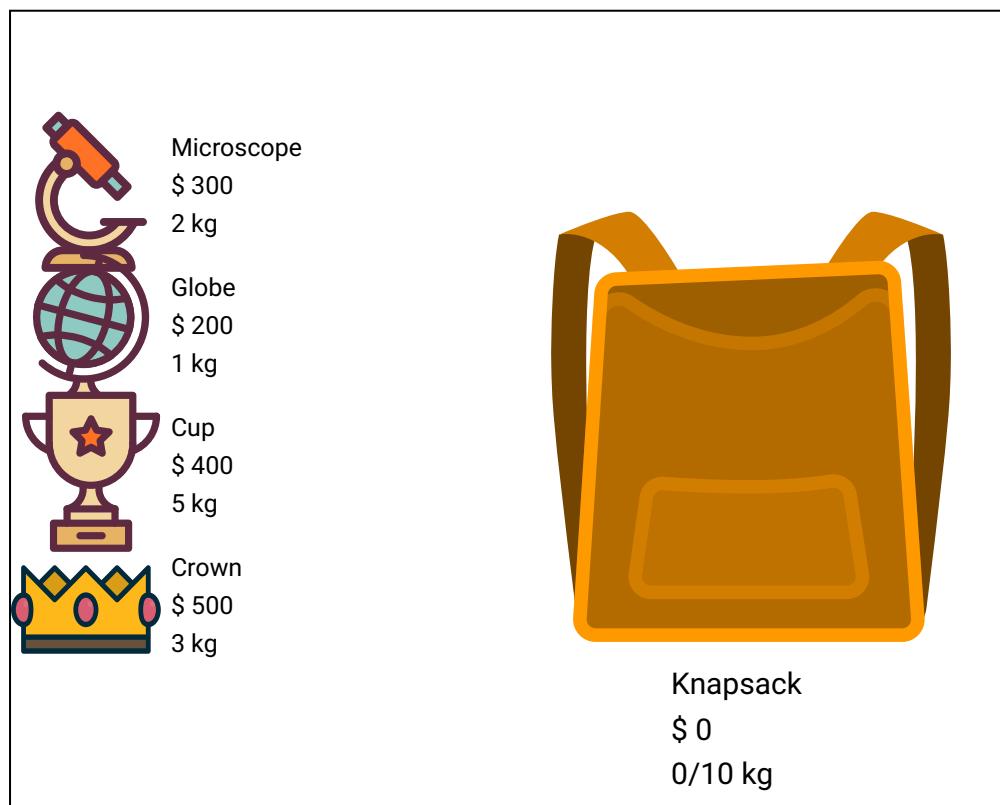
# DSA The 0/1 Knapsack Problem

[« Previous](#)[Next »](#)

## The 0/1 Knapsack Problem

The 0/1 Knapsack Problem states that you have a backpack with a weight limit, and you are in a room full of treasures, each treasure with a value and a weight.

To solve the 0/1 Knapsack Problem you must figure out which treasures to pack to maximize the total value, and at the same time keeping below the backpack's weight limit.



Are you able to solve the 0/1 Knapsack Problem above manually? Continue reading to see different implementations that solves the 0/1 Knapsack Problem.

## The 0/1 Knapsack Problem

### Rules:

- Every item has a weight and value.
- Your knapsack has a weight limit.
- Choose which items you want to bring with you in the knapsack.
- You can either take an item or not, you cannot take half of an item for example.

### Goal:

- Maximize the total value of the items in the knapsack.

## The Brute Force Approach

Using brute force means to just check all possibilities, looking for the best result. This is usually the most straight forward way of solving a problem, but it also requires the most calculations.

To solve the 0/1 Knapsack Problem using brute force means to:

1. Calculate the value of every possible combination of items in the knapsack.
2. Discard the combinations that are heavier than the knapsack weight limit.
3. Choose the combination of items with the highest total value.

### **How it works:**

1. Consider each item one at a time.
  - a. If there is capacity left for the current item, add it by adding its value and reducing the remaining capacity with its weight. Then call the function on itself for the next item.
  - b. Also, try not adding the current item before calling the function on itself for the next item.
2. Return the maximum value from the two scenarios above (adding the current item, or not adding it).

Solving the 0/1 Knapsack Problem using recursion and brute force:

```
1 | def knapsack_brute_force(capacity, n):  
2 |     # Base Case  
3 |     if n == 0 or capacity == 0:  
4 |         return 0  
5 |     else:  
6 |         # Recursive Case  
7 |         include_item = values[n-1] + knapsack_brute_force(capacity - weights[n-1], n-1)  
8 |         exclude_item = knapsack_brute_force(capacity, n-1)  
9 |  
10|         return max(include_item, exclude_item)  
11|  
12| if __name__ == "__main__":  
13|     values = [300, 200, 400, 500]  
14|     weights = [2, 1, 5, 3]  
15|     capacity = 10  
16|     n = len(values)  
17|  
18|     print("\nMaximum value in Knapsack =", knapsack_brute_force(capacity, n))
```

[Run Example »](#)

Running the code above means that the `knapsack_brute_force` function is called many times recursively. You can see that from all the printouts.

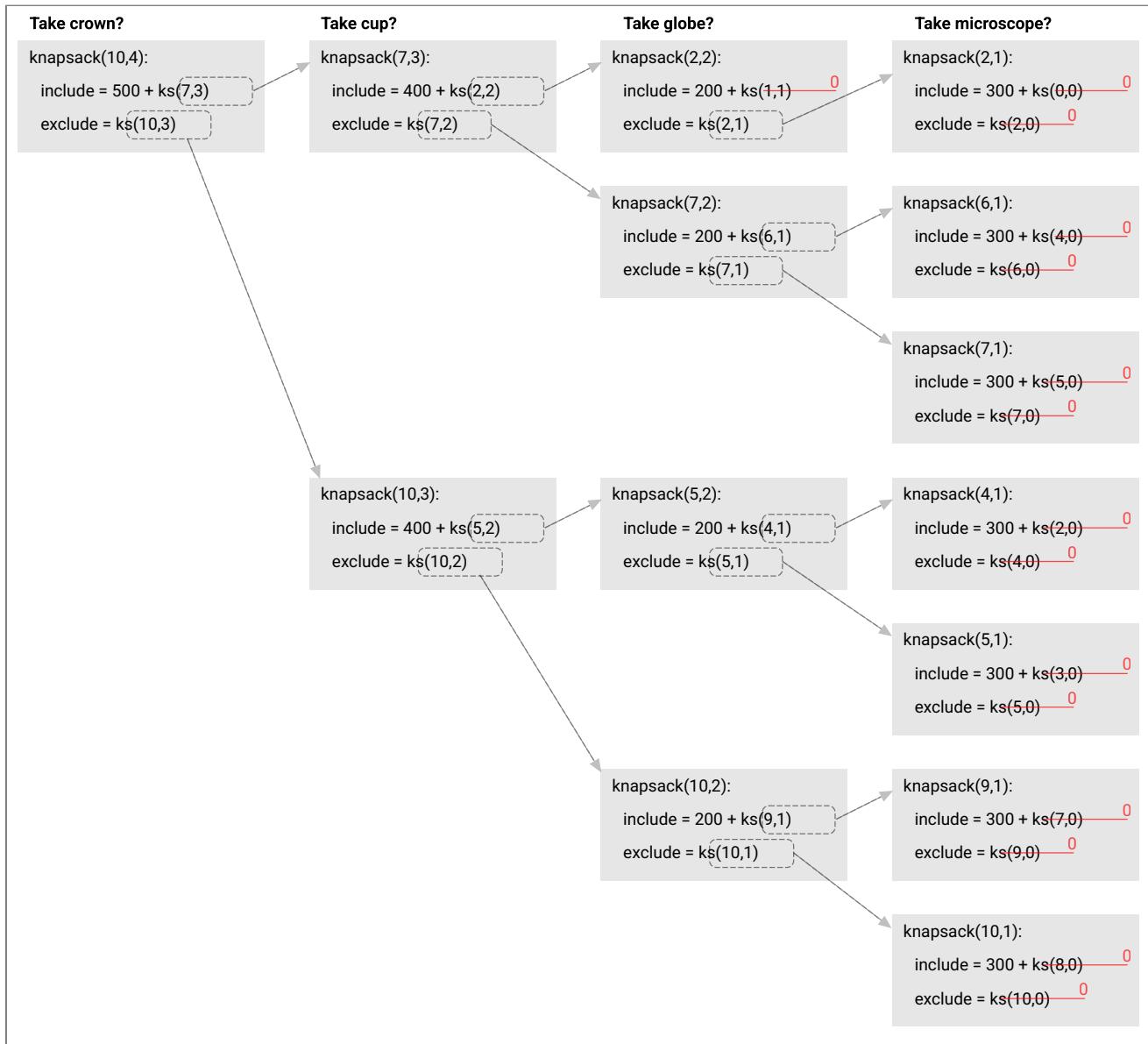
Every time the function is called, it will either include the current item `n-1` or not.

**Line 2:** This print statement shows us each time the function is called.

**Line 3-4:** If we run out of items to check (`n == 0`), or we run out of capacity (`capacity == 0`), we do not do any more recursive calls because no more items can be added to the knapsack at this point.

**Line 6-7:** If the current item is heavier than the capacity (`weights[n-1] > capacity`), forget the current item and go to the next item.

function call:



**Note:** In the recursion tree above, writing the real function name

`knapsack_brute_force(7, 3)` would make the drawing too wide, so "ks(7,3)" or "knapsack(7,3)" is written instead.

From the recursion tree above, it is possible to see that for example taking the crown, the cup, and the globe, means that there is no space left for the microscope (2 kg), and that gives us a total value of  $200+400+500=1100$ .

We can also see that only taking the microscope gives us a total value of 300 (right bottom gray box).

# The Memoization Approach (top-down)

The memoization technique stores the previous function call results in an array, so that previous results can be fetched from that array and does not have to be calculated again.

Read more about memoization [here](#).

Memoization is a 'top-down' approach because it starts solving the problem by working its way down to smaller and smaller subproblems.

In the brute force example above, the same function calls happen only a few times, so the effect of using memoization is not so big. But in other examples with far more items to choose from, the memoization technique would be more helpful.

## How it works:

1. In addition to the initial brute force code above, create an array `memo` to store previous results.
2. For every function call with arguments for capacity `c` and item number `i`, store the result in `memo[c, i]`.
3. To avoid doing the same calculation more than once, every time the function is called with arguments `c` and `i`, check first if the result is already stored in `memo[c, i]`.

After improving the brute force implementation with the use of memoization, the code now looks like this:

## Example

Improved solution to the 0/1 Knapsack Problem using memoization:

```
def knapsack_memoization(capacity, n):
    print(f"knapsack_memoization({n}, {capacity})")
    if memo[n][capacity] is not None:
        print(f"Using memo for ({n}, {capacity})")
        return memo[n][capacity]
```

```
TLIN SASS VUE DSA GEN AI SCIPY AWS CYBERSECURITY DATA SC
10     result = knapsack_memoization(capacity, n-1)
11 else:
12     include_item = values[n-1] + knapsack_memoization(capacity
13     exclude_item = knapsack_memoization(capacity, n-1)
14     result = max(include_item, exclude_item)
15
16     return result
17
18
19 values = [300, 200, 400, 500]
20 weights = [2, 1, 5, 3]
21 capacity = 10
22 n = len(values)
23
24
25
26 print("\nMaximum value in Knapsack =", knapsack_memoization(capaci
```

[Run Example »](#)

The highlighted lines in the code above show the memoization technique used to improve the previous brute force implementation.

**Line 24:** Create an array `memo` where previous results are stored.

**Line 3-5:** At the start of the function, before doing any calculations or recursive calls, check if the result has already been found and stored in the `memo` array.

**Line 16:** Store the result for later.

## The Tabulation Approach (bottom-up)

Another technique to solve the 0/1 Knapsack problem is to use something called tabulation. This approach is also called the iterative approach, and is a technique used in Dynamic Programming.

Tabulation solves the problem in a bottom-up manner by filling up a table with the results from the most basic subproblems first. The next table values are filled in using the previous results.

1. Consider one item at a time, and increase the knapsack capacity from 0 to the knapsack limit.
2. If the current item is not too heavy, check what gives the highest value: adding it, or not adding it. Store the maximum of these two values in the table.
3. In case the current item is too heavy to be added, just use the previously calculated value at the current capacity where the current item was not considered.

Use the animation below to see how the table is filled cell by cell using previously calculated values until arriving at the final result.

### Find the maximum value in the knapsack.

1. Click "Run" to fill the table.
2. After the table is filled, click a cell value to see the calculation.

Weights (kg)	Knapsack capacities (kg)									
	0	1	2	3	4	5	6	7	8	9
	0	0	0	0	0	0	0	0	0	0
	0									
	0									
	0									
	0									
300	200	400	500							
Values (\$)										

Maximum Value in Knapsack:

Speed:

The tabulation approach works by considering one item at a time, for increasing knapsack capacities. In this way the solution is built up by solving the most basic subproblems first.

On each row an item is considered to be added to knapsack, for increasing capacities.

```
1  def knapsack_tabulation():
2      n = len(values)
3      tab = [[0]*(capacity + 1) for y in range(n + 1)]
4
5      for i in range(1, n+1):
6          for w in range(1, capacity+1):
7              if weights[i-1] > w:
8                  tab[i][w] = tab[i-1][w]
9              else:
10                  tab[i][w] = max(tab[i-1][w], values[i-1] + tab[i-1][w - weights[i-1]])
11
12      print(tab)
13
14      return tab[n][capacity]
```

[Run Example »](#)

**Line 7-10:** If the item weight is lower than the capacity it means it can be added. Check if adding it gives a higher total value than the result calculated in the previous row, which represents not adding the item. Use the highest (`max`) of these two values. In other words: Choose to take, or not to take, the current item.

**Line 8:** This line might be the hardest to understand. To find the value that corresponds to adding the current item, we must use the current item's value from the `values` array. But in addition, we must reduce the capacity with the current item's weight, to see if the remaining capacity can give us any additional value. This is similar to check if other items can be added in addition to the current item, and adding the value of those items.

**Line 12:** In case the current item is heavier than the capacity (too heavy), just fill in the value from the previous line, which represents not adding the current item.

corresponding table cell in the animation above to get a better understanding as you read.

**Microscope, capacity 1 kg:** For the first value calculated, it is checked whether the microscope can be put in the bag if the weight limit is 1 kg. The microscope weighs 2 kg, it is too heavy, and so the value 0 is just copied from the cell above which corresponds to having no items in the knapsack. Only considering the microscope for a bag with weight limit 1 kg, means we cannot bring any items and we must leave empty handed with a total value of \$ 0.

**Microscope, capacity 2 kg:** For the second value calculated, we are able to fit the microscope in the bag for a weight limit of 2 kg, so we can bring it, and the total value in the bag is \$ 300 (the value of the microscope). And for higher knapsack capacities, only considering the microscope, means we can bring it, and so all other values in that row is \$ 300.

**Globe, capacity 1 kg:** Considering the globe at 1 kg and a knapsack capacity at 1 kg means that we can bring the globe, so the value is \$ 200. The code finds the maximum between bringing the globe which gives us \$ 200, and the previously calculated value at 1 kg capacity, which is \$ 0, from the cell above. In this case it is obvious that we should bring the globe because that is the only item with such a low weight, but in other cases the previously calculated value at the same capacity might be higher.

**Globe, capacity 2 kg:** At capacity 2 kg, the code sees that the globe can fit, which gives us a value of \$ 200, but then the microscope cannot fit. And adding the microscope for a capacity of 2 kg gives us a value of \$ 300, which is higher, so taking the microscope (value from the cell above) is the choice to maximize knapsack value for this table cell.

**Globe, capacity 3 kg:** Considering the globe with a capacity of 3 kg, means that we can take the globe, but with the remaining capacity of 2 kg we can also take the microscope. In this cell, taking both the globe and the microscope gives us a higher value  $200+300=500$  than taking just the microscope (as calculated on the previous line), so both items are taken and the cell value is 500.

## Which Items Gives Us The Highest Value?

After filling out the table and finding the maximum value the knapsack can have, it is not obvious which items we need to pack with us to get that value.

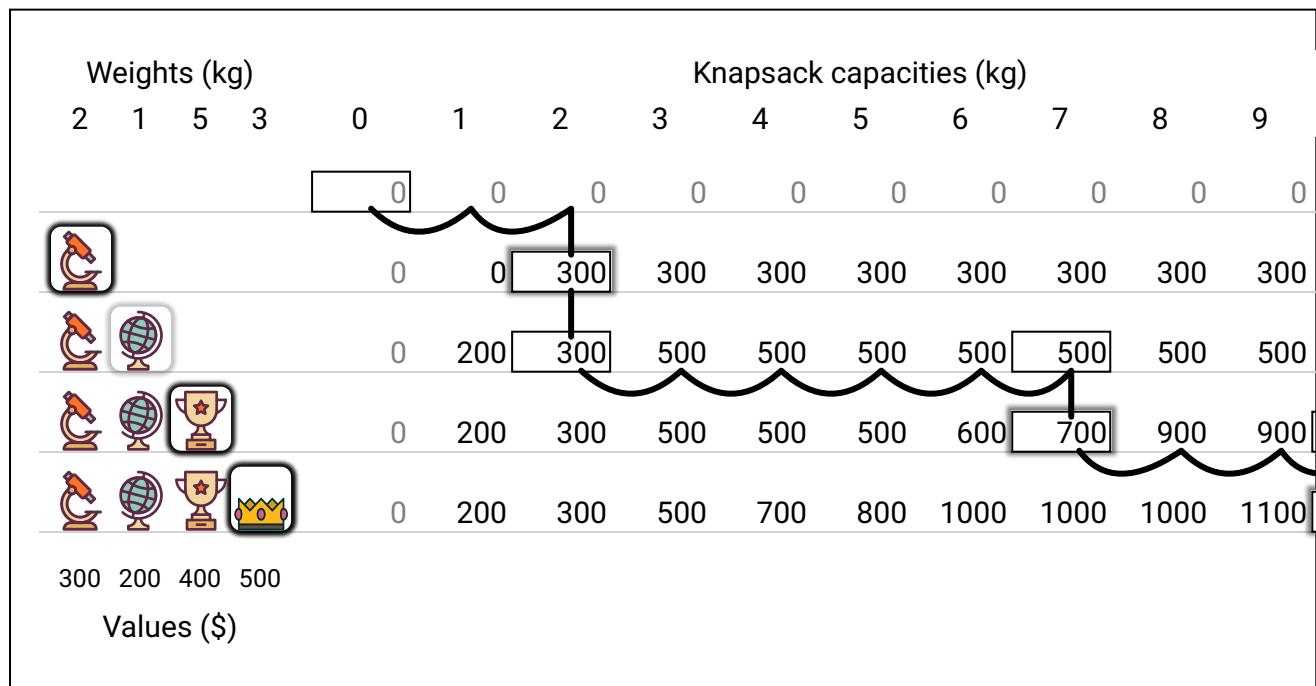
To find the included items, we use the table we have created, and we start with the bottom right cell with the highest value, in our case the cell with value 1200 in it.

**Steps to find the included items:**

we move to the row above, and we move to the left as many times as the weight of the included item.

- Continue to do steps 2 and 3 until a cell with value 0 is found.

Here is a drawing of how the included items are found, using the step-by-step method:



This is how the included items are found:

- The bottom right value is 1200, and the cell above is 900. The values are different, which means the crown is included.
- The next cell we go to is on the row above, and we move left as many times as the crown is heavy, so 3 places left to the cell with value 700.
- The cell we are in now has value 700, and the cell above has value 500. The values are different, which means the item on the current row is included: the cup.
- The cup weighs 5 kg, so the next cell we go to is on the row above, and 5 places to the left, to the cell with value 300, on the row where the globe is considered.
- The cell above has the same value 300, which means the globe is not included, and the next cell we go to is the cell right above with value 300 where the microscope is considered.
- Since the cell above is different than the current cell with value 300, it means the microscope is included.
- The next cell we go to is on the line above, and two places to the left because the microscope is 2 kg.
- We arrive at the upper leftmost cell. Since the value is 0, it means we are finished.

Knapsack problem.

## Example

Extended solution to the 0/1 Knapsack Problem to find the included items:

```
1  def knapsack_tabulation():
2      n = len(values)
3      tab = [[0] * (capacity + 1) for _ in range(n + 1)]
4
5      for i in range(1, n + 1):
6          for w in range(1, capacity + 1):
7              if weights[i-1] <= w:
8                  include_item = values[i-1] + tab[i-1][w - weights[9
9                      exclude_item = tab[i-1][w]
10                     tab[i][w] = max(include_item, exclude_item)
11                 else:
12                     tab[i][w] = tab[i-1][w]
13
14             for row in tab:
15                 print(row)
16
17
18
19
20
21
22
23
24
25         return tab[n][capacity]
26
27
28 values = [300, 200, 400, 500]
29 weights = [2, 1, 5, 3]
30 capacity = 10
31 print("\nMaximum value in Knapsack =", knapsack_tabulation())
```

[Run Example »](#)

The three approaches to solving the 0/1 Knapsack Problem run differently, and with different time complexities.

**Brute Force Approach:** This is the slowest of the three approaches. The possibilities are checked recursively, with the time complexity  $O(2^n)$ , where  $n$  is the number of potential items we can pack. This means the number of computations double for each extra item that needs to be considered.

**Memoization Approach:** Saves computations by remembering previous results, which results in a better time complexity  $O(n \cdot C)$ , where  $n$  is the number of items, and  $C$  is the knapsack capacity. This approach runs otherwise in the same recursive way as the brute force approach.

**Tabulation Approach:** Has the same time complexity as the memoization approach  $O(n \cdot C)$ , where  $n$  is the number of items, and  $C$  is the knapsack capacity, but memory usage and the way it runs is more predictable, which normally makes the tabulation approach the most favorable.

**Note:** Memoization and tabulation are used in something called dynamic programming, which is a powerful technique used in computer science to solve problems. To use dynamic programming to solve a problem, the problem must consist of overlapping subproblems, and that is why it can be used to solve the 0/1 Knapsack Problem, as you can see above in the memoization and tabulation approaches.

[« Previous](#)[Next »](#)

**W3schools Pathfinder**  
Track your progress - it's free!

[Sign Up](#) [Log in](#)

# Memoization

[« Previous](#)[Next »](#)

## Memoization

Memoization is a technique where results are stored to avoid doing the same computations many times.

When Memoization is used to improve recursive algorithms, it is called a "top-down" approach because of how it starts with the main problem and breaks it down into smaller subproblems.

Memoization is used in [Dynamic Programming](#).

## Using Memoization To Find The $n$ th Fibonacci Number

The  $n$ th Fibonacci number can be found using recursion. Read more about how that is done on [this page](#).

The problem with this implementation is that the number of computations and recursive calls "explodes" when trying to find a higher Fibonacci number, because the same computations are done over and over again.

## Example

Find the 6th Fibonacci number with recursion:

☰ TLIN SASS VUE DSA GEN AI SCIPY AWS CYBERSECURITY DATA SC

```
    return n
else:
    return F(n - 1) + F(n - 2)

print('F(6) = ', F(6))
```

[Run Example »](#)

As you can see from running the example above, there are 25 computations, with the same computations done many times, even for just finding the 6th Fibonacci number.

But using memoization can help finding the  $n$ th Fibonacci number using recursion much more effectively.

We use memoization by creating an array `memo` to hold the Fibonacci numbers, so that Fibonacci number `n` can be found as element `memo[n]`. And we only compute the Fibonacci number if it does not already exist in the `memo` array.

## Example

Find the 6th Fibonacci number with recursion, but using memoization to avoid unnecessary recursive calls:

```
def F(n):
    if memo[n] != None: # Already computed
        return memo[n]
    else: # Computation needed
        print('Computing F('+str(n)+')')
        if n <= 1:
            memo[n] = n
        else:
            memo[n] = F(n - 1) + F(n - 2)
    return memo[n]

memo = [None]*7
print('F(6) = ', F(6))
print('memo = ', memo)
```

As you can see by running the examples above, memoization is very helpful to reduce the number of computations.

The number of computations is reduced from 25 in the initial code, to just 7 in the last example using memoization, and the benefit of using memoization increases really fast by how high the Fibonacci number we want to find is.

Finding the 30th Fibonacci number requires 2,692,537 computations in the initial code, but it just requires 31 computations in the algorithm implemented using memoization!

You get this result by running the code below.

## Example

See the difference in calculations for finding the 30th Fibonacci number, with and without memoization:

```
computation_count = 0
def F(n):
    global computation_count
    computation_count += 1
    if n <= 1:
        return n
    else:
        return F(n - 1) + F(n - 2)

computation_count_mem = 0
def F_mem(n):
    if memo[n] != None: # Already computed
        return memo[n]
    else: # Computation needed
        global computation_count_mem
        computation_count_mem += 1
        if n <= 1:
            memo[n] = n
        else:
            memo[n] = F_mem(n - 1) + F_mem(n - 2)
    return memo[n]

print('F(30) = ', F(30))
print(f'Number of computations: {computation_count}')
```

[Run Example »](#)

## Memoization in AVL Trees

For more details about what an AVL Tree is, please see [this page](#).

An AVL tree is a type of Binary Tree that is self-balancing.

Every time a node is inserted or deleted from an AVL tree, the balancing factor must be calculated for all ancestors, using the height of the left and right subtrees to find out if a rotation is needed to restore balance.

To avoid calculating the height of each node (going all the way down to the leaf nodes) to calculate the balancing factors, each node has its subtree height stored.

## Example

```
class TreeNode:  
    def __init__(self, data):  
        self.data = data  
        self.left = None  
        self.right = None
```

[Run Example »](#)

This means that to find the balance factor for a node, the already stored left child's height is subtracted from the already stored right child's height, no other calculations needed.

Storing height in AVL trees is a form of memoization, because values are stored to avoid re-calculating them. In AVL trees, when the height is stored like this, it is called an **augmented property**.

# Tabulation

[« Previous](#)[Next »](#)

## Tabulation

Tabulation is a technique used to solve problems.

Tabulation uses a table where the results to the most basic subproblems are stored first. The table then gets filled with more and more subproblem results until we find the result to the complete problem that we are looking for.

The tabulation technique is said to solve problems "bottom-up" because of how it solves the most basic subproblems first.

Tabulation is a technique used in [Dynamic Programming](#), which means that to use tabulation, the problem we are trying to solve must consist of overlapping subproblems.

## Using Tabulation To Find The $n$ th Fibonacci Number

[The Fibonacci numbers](#) are great for demonstrating different programming techniques, also when demonstrating how tabulation works.

Tabulation uses a table that is filled with the lowest Fibonacci numbers  $F(0) = 0$  and  $F(1) = 1$  first (bottom-up). The next Fibonacci number to be stored in the table is  $F(2) = F(1) + F(0)$ .

The next Fibonacci number is always the sum of the two previous numbers:

$$F(n) = F(n - 1) + F(n - 2)$$

## Example

Finding the 10th Fibonacci number using tabulation:

```
def fibonacci_tabulation(n):
    if n == 0: return 0
    elif n == 1: return 1

    F = [0] * (n + 1)
    F[0] = 0
    F[1] = 1

    for i in range(2, n + 1):
        F[i] = F[i - 1] + F[i - 2]

    print(F)
    return F[n]

n = 10
result = fibonacci_tabulation(n)
print(f"\nThe {n}th Fibonacci number is {result}")
```

[Run Example »](#)

Other ways to find the  $n$ th Fibonacci number include [recursion](#), or the improved version of it using [memoization](#).

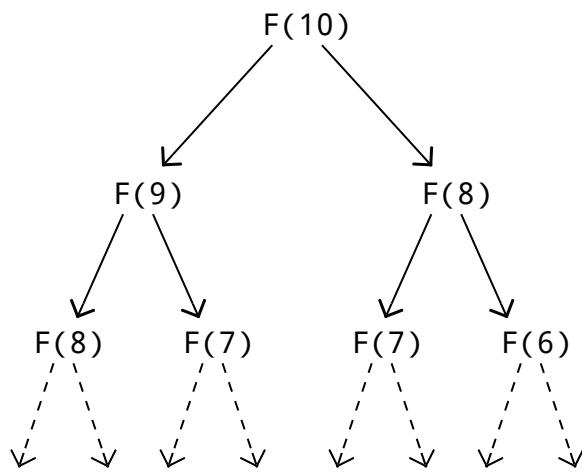
## Tabulation Is A Bottom Up Approach

See the drawings below to get a better idea of why tabulation is called a "bottom up" approach.

As a reference to compare with, see the drawing of the ["top-down" recursion approach](#) to finding the  $n$ th Fibonacci number.



*The bottom up tabulation approach to finding the 10th Fibonacci number.*



*The top down recursive approach to finding the 10th Fibonacci number.*

The tabulation approach starts building the table bottom up to find the 10th Fibonacci number, starting with  $F(0)$  and  $F(1)$ .

The recursive approach start by trying to find  $F(10)$ , but to find that it must call  $F(9)$  and  $F(8)$ , and so it goes all the way down to  $F(0)$  and  $F(1)$  before the function calls start returning values that can be put together to the final answer.

Just like finding the  $n$ th Fibonacci number, tabulation can also be used to find the solution to other problems:

- **The 0/1 Knapsack Problem** is about having a set of items we can pack in a knapsack (a simple backpack), each item with a different value. To solve the problem we need to find the items that will maximize the total value of items we pack, but we cannot bring all the items we want because the knapsack has a weight limit.
- **The Shortest Path Problem** can be solved using [the Bellman-Ford algorithm](#), which also uses tabulation to find the shortest paths in a graph. More specifically, the tabulation approach of the Bellman-Ford algorithm is in how the values in the "distances" array gets updated.
- **The Traveling Salesman Problem** can be solved precisely using the Held-Karp algorithm, which also uses tabulation. This algorithm is not described in this tutorial as it is although better than brute force  $O(n!)$ , still not very effective  $O(2^n n^2)$ , and quite advanced.

## Tabulation in Dynamic Programming

As mentioned in the top, tabulation (just like memoization) is a technique used in something called [Dynamic Programming](#).

Dynamic Programming is a way of designing algorithms to solve problems.

For Dynamic Programming to work, the problem we want to solve must have these two properties:

- The problem must be built up by smaller, **overlapping subproblems**. For example, the solution to Fibonacci number  $F(3)$  overlaps with the solutions to Fibonacci numbers  $F(2)$  and  $F(1)$ , because we get  $F(3)$  by combining  $F(2)$  and  $F(1)$ .
- The problem must also have an **optimal substructure**, meaning that the solution to the problem can be constructed from the solutions to its subproblems. When finding the  $n$ th Fibonacci number,  $F(n)$  can be found by adding  $F(n - 1)$  and  $F(n - 2)$ . So knowing the two previous numbers is not enough to find  $F(n)$ , we must also know the structure so that we know how to put them together.

Read more about Dynamic Programming on the [next page](#).

# Dynamic Programming

[« Previous](#)[Next »](#)

## Dynamic Programming

Dynamic Programming is a method for designing algorithms.

An algorithm designed with Dynamic Programming divides the problem into subproblems, finds solutions to the subproblems, and puts them together to form a complete solution to the problem we want to solve.

To design an algorithm for a problem using Dynamic Programming, the problem we want to solve must have these two properties:

- **Overlapping Subproblems:** Means that the problem can be broken down into smaller subproblems, where the solutions to the subproblems are overlapping. Having subproblems that are overlapping means that the solution to one subproblem is part of the solution to another subproblem.
- **Optimal Substructure:** Means that the complete solution to a problem can be constructed from the solutions of its smaller subproblems. So not only must the problem have overlapping subproblems, the substructure must also be optimal so that there is a way to piece the solutions to the subproblems together to form the complete solution.

We have already seen Dynamic Programming in this tutorial, in the [memoization](#) and [tabulation](#) techniques, and for solving problems like the [0/1 Knapsack Problem](#), or to find [the shortest path](#) with [the Bellman-Ford algorithm](#).

# Using Dynamic Programming To Find The $n$ th Fibonacci Number

Let's say we want an algorithm that finds the  $n$ th Fibonacci number. We don't know how to find the  $n$ th Fibonacci number yet, except that we want to use Dynamic Programming to design the algorithm.

The Fibonacci numbers is a sequence of numbers starting with 0 and 1, and the next numbers are created by adding the two previous numbers.

The 8 first Fibonacci numbers are: 0, 1, 1, 2, 3, 5, 8, 13.

And counting from 0, the 4th Fibonacci number  $F(4)$  is 3.

In general, this is how a Fibonacci number is created based on the two previous:

$$F(n) = F(n - 1) + F(n - 2)$$

So how can we use Dynamic Programming to design an algorithm that finds the  $n$ th Fibonacci number?

There is no exact rule for how to design an algorithm using Dynamic Programming, but here is a suggestion that should work in most cases:

1. Check if the problem has "overlapping subproblems" and an "optimal substructure".
2. Solve the most basic subproblems.
3. Find a way to put the subproblem solutions together to form solutions to new subproblems.
4. Write the algorithm (the step-by-step procedure).
5. Implement the algorithm (test if it works).

Let's do it.

Step 1: Check if the problem has "overlapping subproblems" and an "optimal substructure".

Yes. The 6th Fibonacci number is a combination of the 5th and 4th Fibonacci number:  $8 = 5 + 3$ . And this rule holds for all other Fibonacci numbers as well. This shows that the problem of finding the  $n$ th Fibonacci number can be broken into subproblems.

Also, the subproblems overlap because  $F(5)$  is based on  $F(4)$  and  $F(3)$ , and  $F(6)$  is based on  $F(5)$  and  $F(4)$ .

$$\begin{aligned} F(5) &= \underline{F(4)} + F(3) \\ 5 &= \underline{3} + 2 \end{aligned}$$

*and*

$$\begin{aligned} F(6) &= F(5) + \underline{F(4)} \\ 8 &= 5 + \underline{3} \end{aligned}$$

You see? Both solutions to subproblems  $F(5)$  and  $F(6)$  are created using the solution to  $F(4)$ , and there are many cases like that, so the subproblems overlap as well.

### Optimal substructure?

Yes, the Fibonacci number sequence has a very clear structure, because the two previous numbers are added to create the next Fibonacci number, and this holds for all Fibonacci numbers except for the two first. This means we know *how* to put together a solution by combining the solutions to the subproblems.

We can conclude that the problem of finding the  $n$ th Fibonacci number satisfies the two requirements, which means that we can use Dynamic Programming to find an algorithm that solves the problem.

---

## Step 2: Solve the most basic subproblems.

We can now start trying to find an algorithm using Dynamic Programming.

Solving the most basic subproblems first is a good place to start to get an idea of how the algorithm should run.

In our problem of finding the  $n$ th Fibonacci number, finding the most basic subproblems is not that hard, because we already know that

$$F(3) = 4$$

$$F(4) = 3$$

$$F(5) = 5$$

$$F(6) = 8$$

...

## Step 3: Find a way to put the subproblem solutions together to form solutions to new subproblems.

In this step, for our problem, how the subproblems are put together is quite straightforward, we just need to add the two previous Fibonacci numbers to find the next one.

So for example, the 2nd Fibonacci number is created by adding the two previous numbers  $F(2) = F(1) + F(0)$ , and that is the general rule as well, like mentioned earlier:  
 $F(n) = F(n - 1) + F(n - 2)$ .

**Note:** In other problems, combining solutions to subproblems to form new solutions usually involves making decisions like "should we choose this way, or this way?", or "should we include this item, or not?".

## Step 4: Write the algorithm (the step-by-step procedure).

Instead of writing the text for the algorithm straight away, it might be wise to try to write a procedure to solve a specific problem first, like finding the 6th Fibonacci number.

For reference, the 8 first Fibonacci numbers are: 0, 1, 1, 2, 3, 5, 8, 13.

Finding the 6th Fibonacci number, we could start with the two first numbers 0 and 1, which appear in place 0 and 1 in the sequence, and put them in an array, at index 0 and 1. Then we could add the two first numbers in the array to generate the next number, and push that new number as a new element to the array. If we continue like this until the array is 7 elements long we would stop and return `F[6]`. That would work, right?

After solving the specific problem above, it is now easier to write the actual algorithm.

The algorithm for finding the  $n$ th Fibonacci number, using Dynamic Programming as a design method, can be described like this:

1. Create an array  $F$ , with  $n + 1$  elements.
2. Store the two first Fibonacci numbers  $F[0]=0$  and  $F[1]=1$ .
3. Store the next element  $F[2]=F[1]+F[0]$ , and continue to create new Fibonacci numbers like that until the value in  $F[n]$  is created.
4. Return  $F[n]$ .

## Step 5: Implement the algorithm (test if it works).

To implement the algorithm above, we assume that the argument  $n$  to the function is a positive number (the  $n$ th Fibonacci number), we use a `for` loop to create new Fibonacci numbers, and we return the base cases  $F[0]$  and  $F[1]$  straight away if the function is called with `0` or `1` as an argument.

Implementing the algorithm also means that we can check if it works.

## Example

Finding the 6th Fibonacci number with our new algorithm:

```
def nth_fibo(n):
    if n==0: return 0
    if n==1: return 1

    F = [None] * (n + 1)
    F[0] = 0
    F[1] = 1

    for i in range(2, n + 1):
        F[i] = F[i - 1] + F[i - 2]

    return F[n]

n = 6
result = nth_fibo(n)
print(f"The {n}th Fibonacci number is {result}")
```

There it is!

We have used Dynamic Programming as a design method to create an algorithm that finds the  $n$ th Fibonacci number.

We have also implemented the algorithm to demonstrate that it works, and in doing so we have unintentionally used a well established technique within Dynamic Programming called tabulation, where the solution is found by solving subproblems bottom-up, using some kind of table.

Furthermore, we have avoided calculating the same overlapping subproblems many times, like [F\[3\]](#) for example, that we could potentially have ended up doing otherwise, with a brute force recursive approach for example.

Another technique used in Dynamic Programming is called memoization. In this case, using memoization essentially solves the problem recursively with brute force, but stores the subproblem solutions for later as the algorithm runs to avoid doing the same calculations more than once.

---

## Techniques Used in Dynamic Programming

It might be difficult to design an algorithm using Dynamic Programming, but the concept of Dynamic Programming is actually not that hard: Solve the problem, but since the subproblems are overlapping, do it in a smart way so that a specific subproblem only needs to be solved once.

To be able to use solutions to previously solved subproblems in Dynamic Programming, the previously found solutions must be stored somehow, and that can be done using memoization or tabulation.

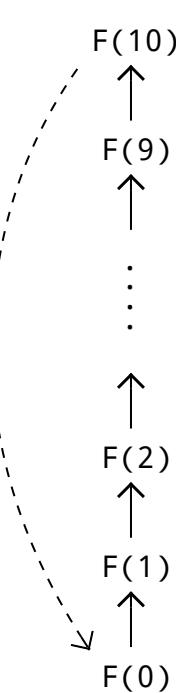
**Memoization** is a technique used in Dynamic Programming, where the solution is found recursively. As the algorithm runs, solutions to subproblems are stored, and before trying to compute the solution to a subproblem, it checks first to see if that solution has already been computed, to avoid doing the same computation more than once.

The memoization technique is called "top-down" because the initial function call is for the main problem, and it results in new function calls for solving smaller and smaller subproblems.

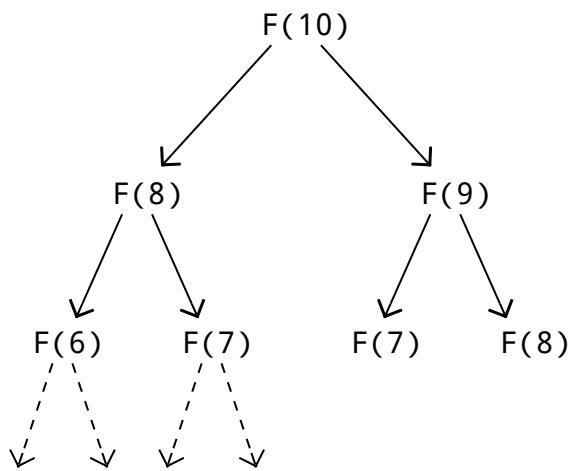
**Tabulation** is a technique used in Dynamic Programming, where solutions to the overlapping subproblems are stored in a table (array), starting with the most basic subproblems.

The tabulation technique is not recursive, and it is called "bottom-up" because of the way the final solution is built up by solving the most basic subproblems first. Since the most basic subproblem solutions are stored in the table first, when solving a subproblem later that relies on previous

tabulation works, and is "bottom-up", take a look at the two images below.



*The bottom-up tabulation approach to finding the 10th Fibonacci number.*



*The top-down memoization approach to finding the 10th Fibonacci number.*

As you can see in the images above, the tabulation approach starts at the bottom by solving  $F(0)$  first, while the memoization approach starts at the top with  $F(10)$  and breaks it into smaller and smaller subproblems from there.

# DSA Greedy Algorithms

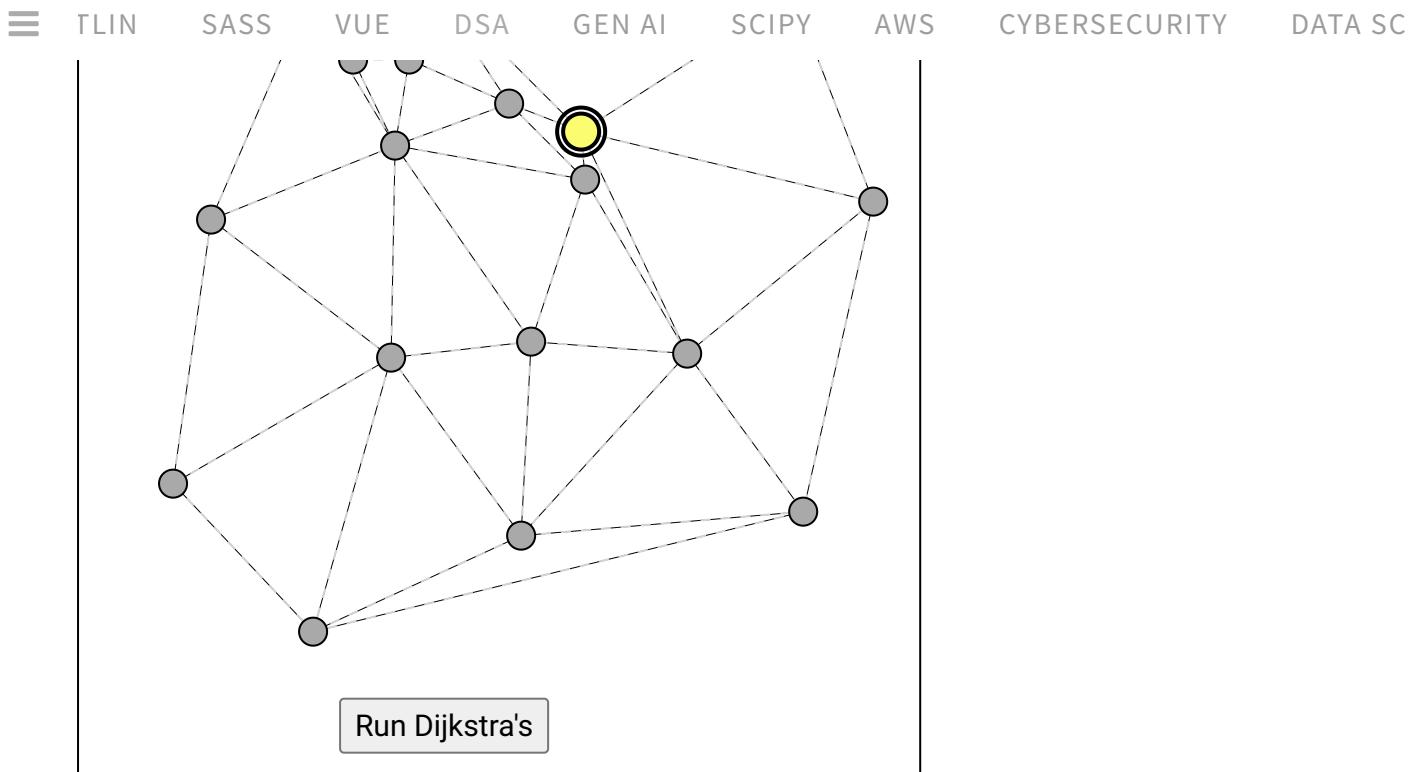
[« Previous](#)[Next »](#)

## Greedy Algorithms

A greedy algorithm decides what to do in each step, only based on the current situation, without a thought of how the total problem looks like.

In other words, a greedy algorithm makes the locally optimal choice in each step, hoping to find the global optimum solution in the end.

In Dijkstra's algorithm for example, the next vertex to be visited is always the next unvisited vertex with the currently shortest distance from the source, as seen from the current group of visited vertices.



So Dijkstra's algorithm is greedy because the choice of which vertex to visit next is only based on the currently available information, without considering the overall problem or how this choice might affect future decisions or the shortest paths in the end.

Choosing a greedy algorithm is a design choice, just like Dynamic Programming is another algorithm design choice.

Two properties must be true for a problem for a greedy algorithm to work:

- **Greedy Choice Property:** Means that the problem is so that the solution (the global optimum) can be reached by making greedy choices in each step (locally optimal choices).
- **Optimal Substructure:** Means that the optimal solution to a problem, is a collection of optimal solutions to sub-problems. So solving smaller parts of the problem locally (by making greedy choices) contributes to the overall solution.

Most of the problems in this tutorial, like sorting an array, or finding the shortest paths in a graph, have these properties, and those problems can therefore be solved by greedy algorithms like Selection sort or Dijkstra's algorithm.

But problems like The Traveling Salesman, or the 0/1 Knapsack Problem, do not have these properties, and so a greedy algorithm can not be used to solve them. These problems are discussed further down.

# Algorithms That Are Not Greedy

Below are algorithms that are not greedy, meaning they do not only rely on doing locally optimal choices in each step:

- **Merge Sort:** Splits the array in halves over and over again, and then merges the array parts together again in a way that results in a sorted array. These operations are not a series of locally optimal choices like greedy algorithms are.
- **Quick Sort:** The choice of pivot element, the arranging of elements around the pivot element, and the recursive calls to do the same with the left and right side of the pivot element – those actions do not rely on making greedy choices.
- **BFS and DFS Traversal:** These algorithms traverse a graph without making a choice locally in each step on how to continue with the traversal, and so they are not greedy algorithms.
- **Finding the nth Fibonacci number using memoization:** This algorithm belongs to a way of solving problems called Dynamic Programming, which solves overlapping sub-problems, and then pieces them back together. Memoization is used in each step to optimize the overall algorithm, which means that at each step, this algorithm does not only consider what is the locally optimal solution, but it also takes into account that a result computed in this step, might be used in later steps.

## The 0/1 Knapsack Problem

The 0/1 Knapsack Problem cannot be solved by a greedy algorithm because it does not fulfill the greedy choice property, and the optimal substructure property, as mentioned earlier.

### **The 0/1 Knapsack Problem**

#### Rules:

- Every item has a weight and value.
- Your knapsack has a weight limit.
- Choose which items you want to bring with you in the knapsack.
- You can either take an item or not, you cannot take half of an item for example.

#### Goal:

- Maximize the total value of the items in the knapsack.

Let's say your backpack's limit is 10 kg, and you have these three treasures in front of you:

Treasure	Weight	Value
An old shield	5 kg	\$300
A nicely painted clay pot	4 kg	\$500
A metal horse figure	7 kg	\$600

Making the greedy choice by taking the most valuable thing first, the horse figure with value \$600, means that you can not bring any of the other things without breaking the weight limit.

So by trying to solve this problem in a greedy way you end up with a metal horse with value \$600.

But the best solution in this case is taking the shield and the pot, maximizing the total value in the backpack to be \$800, while still being under the weight limit.

*What about always taking the treasure with the lowest weight? Or always taking the treasure with the highest value to weight ratio?*

While following those principles would actually lead us to the best solution in this specific case, we could not guarantee that those principles would work if the values and weights in this example were changed.

This means that The 0/1 Knapsack Problem cannot be solved with a greedy algorithm.

Read more about The 0/1 Knapsack Problem [here](#).

## The Traveling Salesman

The Traveling Salesman Problem is a famous problem that also cannot be solved by a greedy algorithm, because it does not fulfill the greedy choice property, and the optimal substructure property, as mentioned earlier.

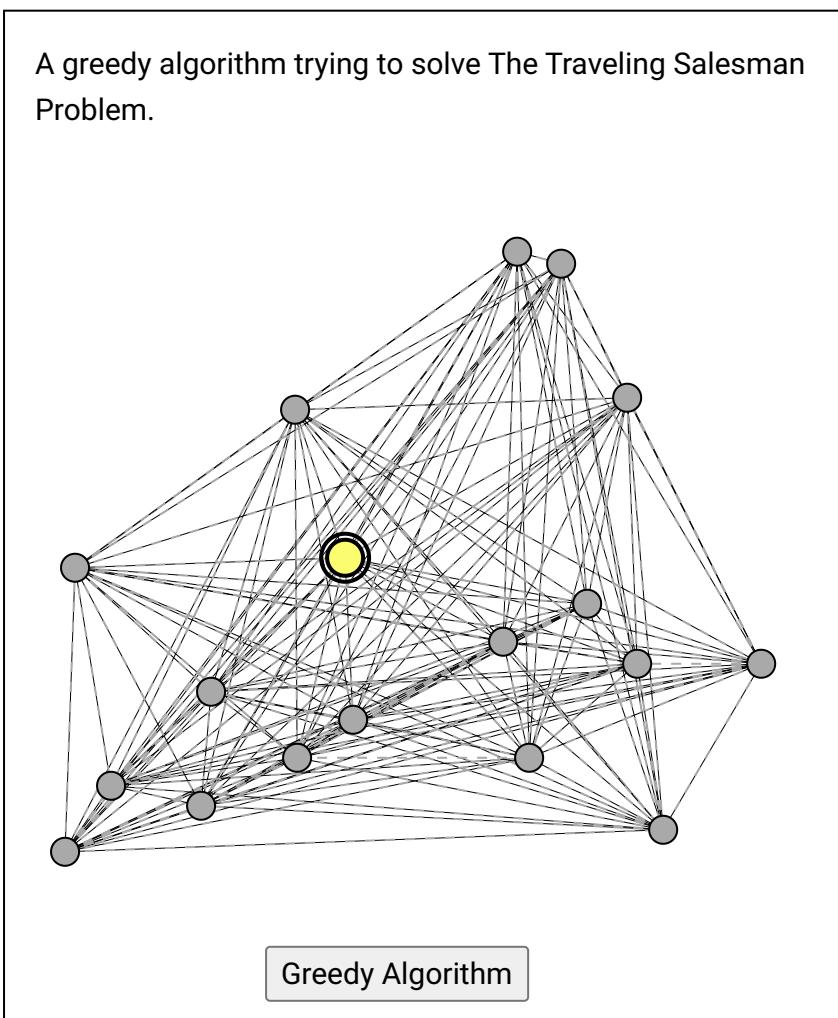
The Traveling Salesman Problem states that you are a salesperson with a number of cities or towns you must visit to sell your things.

### The Traveling Salesman Problem

Using a greedy algorithm here, you would always go to the next unvisited city that is closest to the city you are currently in. But this would in most cases not lead you to the optimal solution with the shortest total path.

The simulation below shows how it looks like when a greedy algorithm tries to solve The Traveling Salesman Problem.

Running the simulation, it might not always be obvious that the algorithm is flawed, but you might notice how sometimes the lines are crossing, creating a longer total distance, when that is clearly not necessary.



While running a greedy approach to The Traveling Salesman Problem sometimes gives us a pretty good approximation to the shortest possible route, a greedy algorithm will not be able to solve The Traveling Salesman Problem in general.

The Traveling Salesman Problem does not fulfill the properties needed so that it can be solved by a greedy algorithm.

Read more about the Traveling Salesman Problem [here](#).

Previous

Next >

W3schools Pathfinder

**Track your progress - it's free!**

[Sign Up](#) [Log in](#)

**Level up your  
skills with our  
courses!**

*Gain skills, and get certified.*

Start Today!



## COLOR PICKER

