13 Recursion

A function that calls itself is said to be **recursive**, and the technique of employing a recursive function is called **recursion**.

Why Use Recurion

some situations particularly lend themselves to a **self-referential** definition. For example, traversal of **tree-like data structures**, these are nested structures, they readily fit a recursive definition.

On the other hand, recursion isn't for every situation. Here are some other factors to consider:

- For some problems, a recursive solution, though possible, will be awkward rather than elegant.
- Recursive implementations often consume more memory than non-recursive ones.
- In some cases, using recursion may result in slower execution time.
  Typically, the readability of the code will be the biggest determining factor. But it depends on the circumstances.

Recursion in Python

When you call a function in Python, the interpreter creates a new **local namespace** so that names defined within that function don't collide with identical names defined elsewhere. One function can call another, and even if they both define objects with the same name, it all works out fine because those objects exist in separate **namespaces**.

The same holds true if multiple instances of the same function are running concurrently.

For example

```
def function():
    x = 10
    function()

>>> function()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in function
  File "<stdin>", line 3, in function
  File "<stdin>", line 3, in function
  [Previous line repeated 996 more times]
RecursionError: maximum recursion depth exceeded
```

When `function()` executes the first time, Python creates a namespace and assigns `x` the value `10` in that namespace. The second time `function()` runs, the interpreter creates a second namespace and assigns `10` to `x` there as well. These two instances of the name `x` are distinct from each another and can coexist without clashing because they are in separate namespaces.

As written, `function()` would in theory go on forever, calling itself over and over without any of the calls ever returning. In practice, of course, nothing is truly forever. Your computer only has so much memory, and it would run out eventually.

Python doesn't allow that to happen. The interpreter limits the maximum number of times a function can call itself recursively, and when it reaches that limit, it raises a `RecursionError` exception, as you see above.

You can find out what Python's recursion limit is with a function from the `sys` module called `getrecursionlimit()`:

```
>>> from sys import getrecursionlimit
>>> getrecursionlimit()
1000
```

You can change it, too, with `setrecursionlimit()`:

```
>>> from sys import setrecursionlimit
>>> setrecursionlimit(2000)
>>> getrecursionlimit()
2000
```

A function that calls itself recursively must have a plan to eventually stop. Recursive functions typically follow this pattern

- There are one or more base cases that are directly solvable without the need for further recursion.
- Each recursive call moves the solution progressively closer to a base case.

Get Started: Count Down to Zero

The first example is a function called `countdown()`, which takes a positive number as an argument and prints the numbers from the specified argument down to zero.

```
# Recursive implementation
>>> def countdown(n):
...     print(n)
...     if n == 0:
...         return             # Terminate recursion
...     else:
...         countdown(n - 1)   # Recursive call
...

>>> countdown(5)
5
4
3
2
1
0
```

Notice how `countdown()` fits the paradigm for a recursive algorithm described above:

- The base case occurs when `n` is zero, at which point recursion stops.
- In the recursive call, the argument is one less than the current value of `n`, so each recursion moves closer to the base case.

Here's one possible non-recursive implementation for comparison

```
>>> def countdown(n):
...     while n >= 0:
...         print(n)
...         n -= 1
...

>>> countdown(5)
5
4
3
2
1
0
```

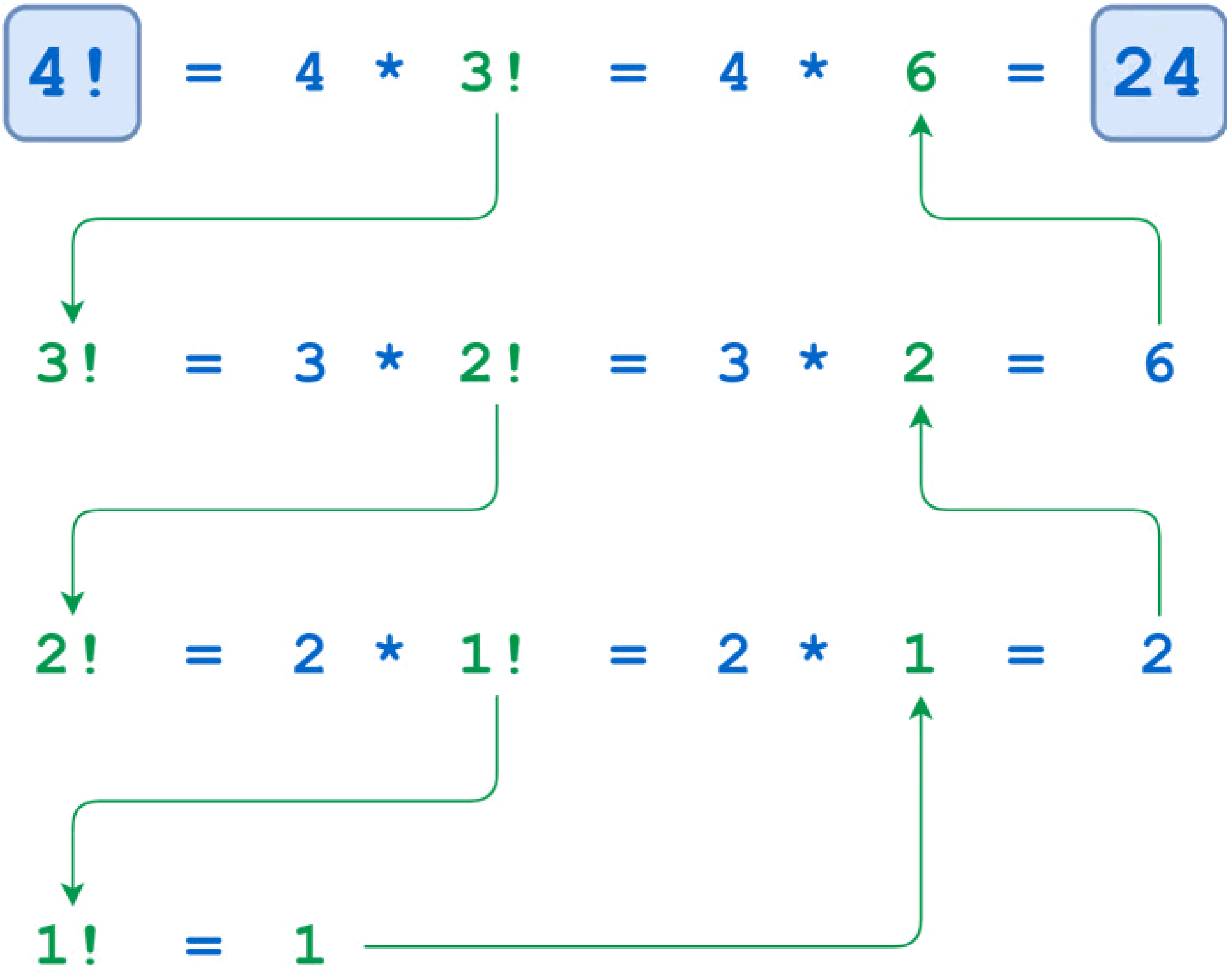
Calculate Factorial

The factorial of a positive integer *n*, denoted as *n*!, is the product of all integers from 1 to *n*, inclusive.

```
n! = 1x2x3x4x...xn
```

There are base cases that are solvable without recursion.

- The base cases (*n* = 0 or *n* = 1) are solvable without recursion.
- For values of *n* greater than 1, *n*! is defined in terms of (*n* - 1)!, so the recursive solution progressively approaches the base case.
  For example, recursive computation of 4! looks like this:



  The calculations of 4!, 3!, and 2! suspend until the algorithm reaches the base case where *n* = 1. At that point, 1! is computable without further recursion, and the deferred calculations run to completion.

Define a Python Factorial Function

Here’s a recursive Python function to calculate factorial.

```
>>> def factorial(n):
...     print(f"factorial() called with n = {n}")
...     return_value = 1 if n <= 1 else n * factorial(n -1)
...     print(f"-> factorial({n}) returns {return_value}")
...     return return_value
...

>>> factorial(4)
factorial() called with n = 4
factorial() called with n = 3
factorial() called with n = 2
factorial() called with n = 1
-> factorial(1) returns 1
-> factorial(2) returns 2
-> factorial(3) returns 6
-> factorial(4) returns 24
24
```

Recursion isn't necessary here. You could implement `factorial()` iteratively using a `for` loop

```
>>> def factorial(n):
...     return_value = 1
...     for i in range(2, n + 1):
...         return_value *= i
...     return return_value
...

>>> factorial(4)
24
```

Speed Comparison of Factorial Implementations

To evaluate execution time, you can use a function called `timeit()` from a module that is also called `timeit`.

```
>>> from timeit import timeit

>>> timeit("print(string)", setup="string='foobar'", number=100)
foobar
foobar
foobar
   .
   . [100 repetitions]
   .
foobar
0.03347089999988384
```

Here, the `setup` parameter assigns `string` the value `'foobar'`.
Then `timeit()` prints `string` one hundred times. The total execution time is just over 3/100 of a second.

The examples shown below use `timeit()` to compare the recursive and iterative implementations of factorial from above.

First, here's the recursive version

```
>>> setup_string = """
... print("Recursive:")
... def factorial(n):
...     return 1 if n <= 1 else n * factorial(n - 1)
... """

>>> from timeit import timeit
>>> timeit("factorial(4)", setup=setup_string, number=10000000)
Recursive:
4.957105500000125
```

Next up is the iterative implementation

```
>>> setup_string = """
... print("Iterative:")
... def factorial(n):
...     return_value = 1
...     for i in range(2, n + 1):
...         return_value *= i
...     return return_value
... """

>>> from timeit import timeit
>>> timeit("factorial(4)", setup=setup_string, number=10000000)
Iterative:
3.733752099999947
```

In Python, you don't need to implement a factorial function at all. It's already available in the standard `math` module.

It might interest you to know how this performs in the timing test

```
>>> setup_string = "from math import factorial"

>>> from timeit import timeit
>>> timeit("factorial(4)", setup=setup_string, number=10000000)
0.3724050999999946
```

`math.factorial()` performs better than the best of the other two implementations shown above.

Detect Palindromes

A **palindrome** is a word that reads the same backward as it does forward. Examples include the following words

- Racecar
- Level
- Kayak
- Reviver
- Civic

Consider this recursive definition of a palindrome

- **Base cases:** An empty string and a string consisting of a single character are inherently palindromic.
- **Reductive recursion:** A string of length two or greater is a palindrome if it satisfies both of these criteria:
  1. The first and last characters are the same.
  2. The substring between the first and last characters is a palindrome.
  Slicing helps here. For a string `word`, indexing and slicing give the following substrings:
- The first character is `word[0]`.
- The last character is `word[-1]`.
- The substring between the first and last characters is `word[1:-1]`.
  So you can define `is_palindrome()` recursively like this

```
>>> def is_palindrome(word):
...     """Return True if word is a palindrome, False if not."""
...     if len(word) <= 1:
...         return True
...     else:
...         return word[0] == word[-1] and is_palindrome(word[1:-1])
...

>>> # Base cases
>>> is_palindrome("")
True
```

```
>>> is_palindrome("a")
True

>>> # Recursive cases
>>> is_palindrome("foo")
False
>>> is_palindrome("racecar")
True
>>> is_palindrome("troglodyte")
False
>>> is_palindrome("civic")
True
```

Sort With Quicksort

Quicksort is a **divide-and-conquer algorithm**. Suppose you have a list of objects to sort. You start by choosing an item in the list, called the **pivot** item. This can be any item in the list. You then **partition** the list into two sublists based on the pivot item and recursively sort the sublists.

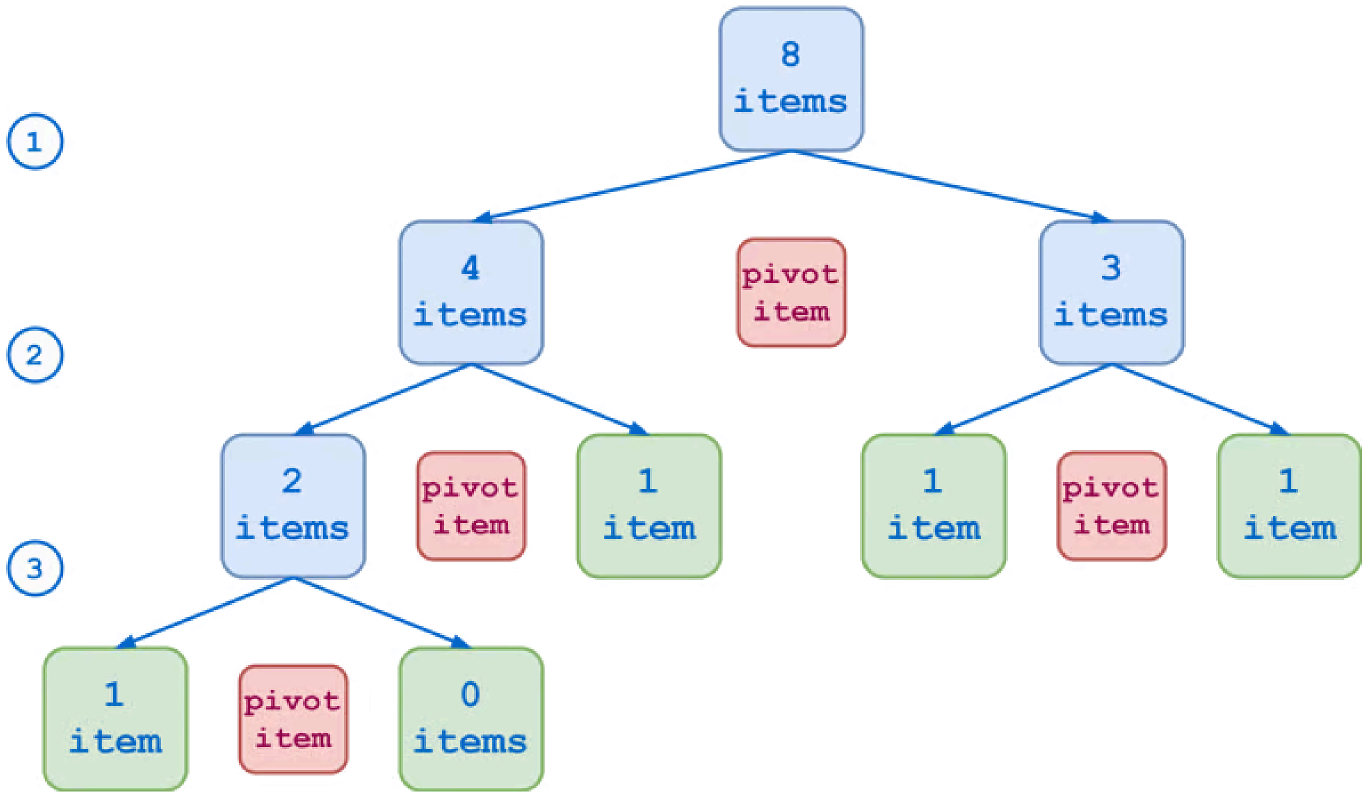
The steps of the algorithm are as follows

- Choose the pivot item.
- Partition the list into two sublists:
  1. Those items that are less than the pivot item
  2. Those items that are greater than the pivot item
- Quicksort the sublists recursively.

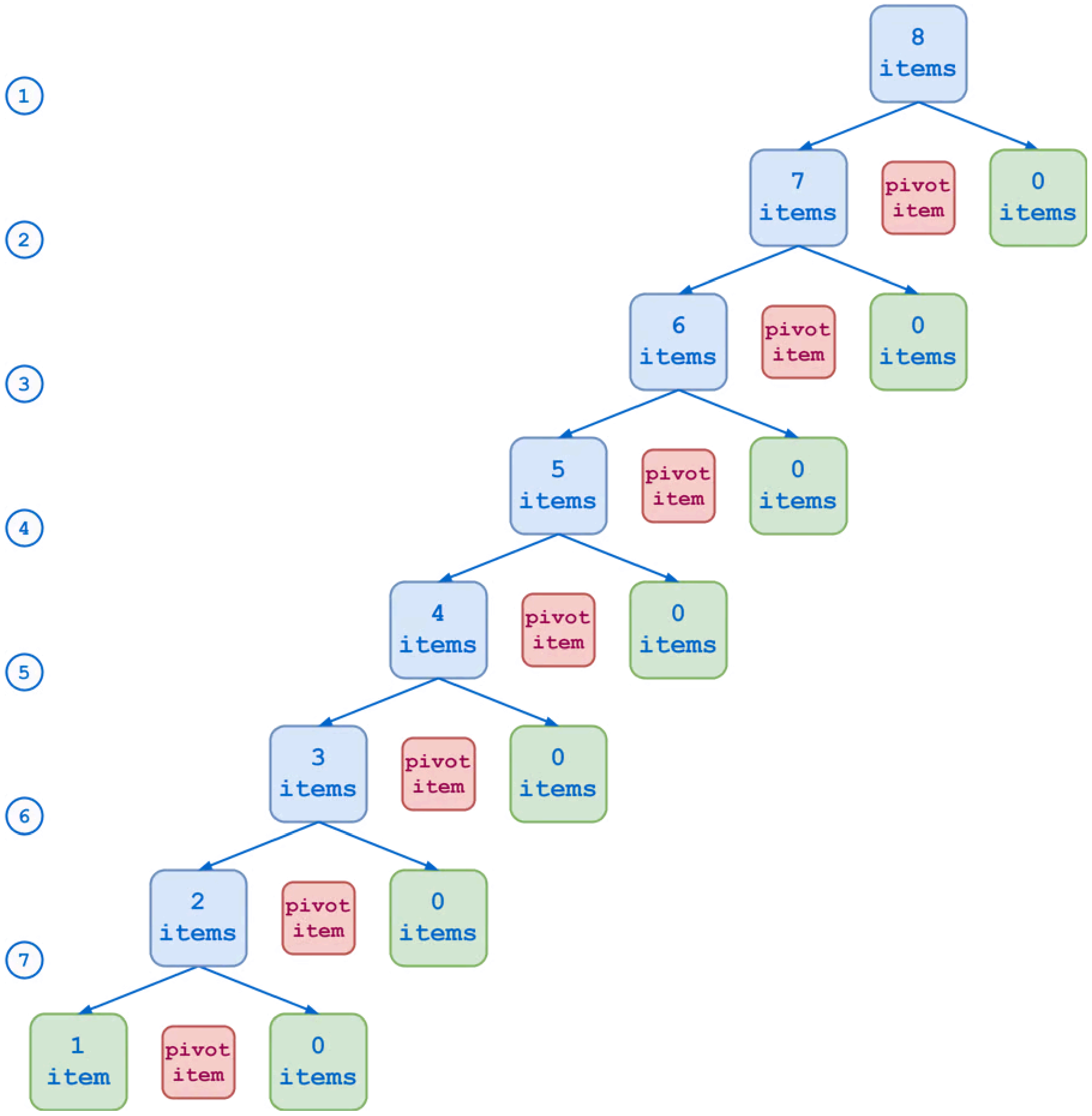
Choosing the Pivot Item

The Quicksort algorithm will work no matter what item in the list is the pivot item. But some choices are better than others.

Imagine that your initial list to sort contains eight items. If each partitioning results in sublists of roughly equal length, then you can reach the base cases in three steps.



If your choice of pivot item is especially unlucky, each partition results in one sublist that contains all the original items except the pivot item and another sublist that is empty. In that case, it takes seven steps to reduce the list to the base cases



The Quicksort algorithm will be more efficient in the first case. But you’d need to know something in advance about the nature of the data you’re sorting in order to systematically choose optimal pivot items.

Implementing the Partitioning

Once you’ve chosen the pivot item, the next step is to partition the list. Again, the goal is to create two sublists, one containing the items that are less than the pivot item and the other containing those that are greater.

You could accomplish this directly in place. In other words, by swapping items, you could shuffle the items in the list around until the pivot item is in the middle, all the lesser items are to its left, and all the greater items are to its right. Then, when you Quicksort the sublists recursively, you'd pass the slices of the list to the left and right of the pivot item.

Alternately, you can use Python's list manipulation capability to create new lists instead of operating on the original list in place.

The algorithm is as follows

- Choose the pivot item using the median-of-three method described above.
- Using the pivot item, create three sublists:
    1. The items in the original list that are less than the pivot item
    2. The pivot item itself
    3. The items in the original list that are greater than the pivot item
- Recursively Quicksort lists 1 and 3.
- Concatenate all three lists back together.

Note that this involves creating a third sublist that contains the pivot item itself. One advantage to this approach is that it smoothly handles the case where the pivot item appears in the list more than once.

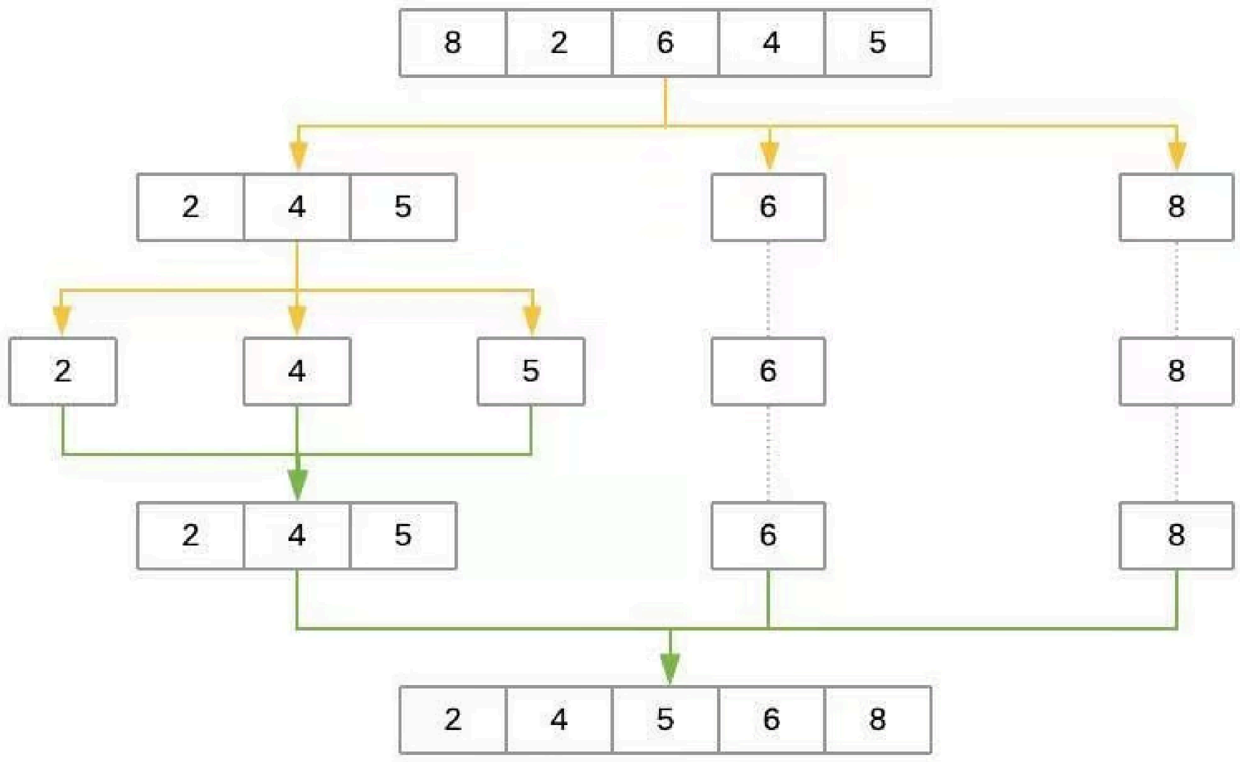
Using the Quicksort Implementation

Here's the Quicksort Python code

```
import statistics

def quicksort(arr):
        if len(arr) <= 1:
                return arr
        else:
                pivot = statistics.median(
                        [
                                arr[0],
                                arr[len(arr)//2]
                                arr[-1]
                        ]
                )
                less, pivot_same, greater = [], [], []
                for i in arr:
                        if n < pivot:
                                less.append(i)
                        elif n == pivot:
                                pivot_same.append(i)
                        elif n > pivot:
                                greater.append(i)
                )
                return (
                        quicksort(less) +
                        pivot_same +
                        quicksort(greater)
                )

arr = [randint(0, 1000) for i in range(16)]
print(arr)
quicksort(arr)
print(arr)
```



Note: This example has the advantage of being succinct and relatively readable. However, it isn’t the most efficient implementation.