Quick Sort Algorithm Using Python

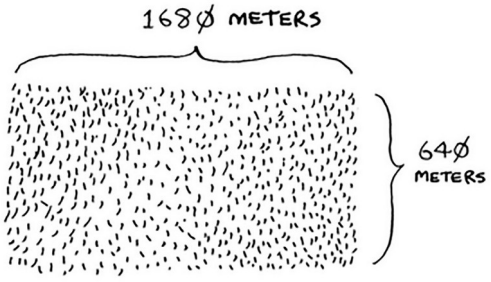
part1: Divide & conquer

1 . sometimes you'll come across a problem that can't be solved by any algorithm you've learned. When a good algorithm it comes across such a problem, they don't just give up. They have a toolbox full of techniques they use on the problem, trying to come up with a solution. Divide-and conquer is the first general technique you learn.

2 . D&C gives you a new way to think about solving problems. D&C is another tool in your toolbox. When you get a new
problem, you don't have to be stumped. Instead, you can ask, "Can I solve this if I use divide and conquer?"
At the end of the chapter, you'll learn your first major D&C algorithm: quicksort. Quicksort is a sorting algorithm, and a much faster one than selection sort . It's a good example of elegant code.

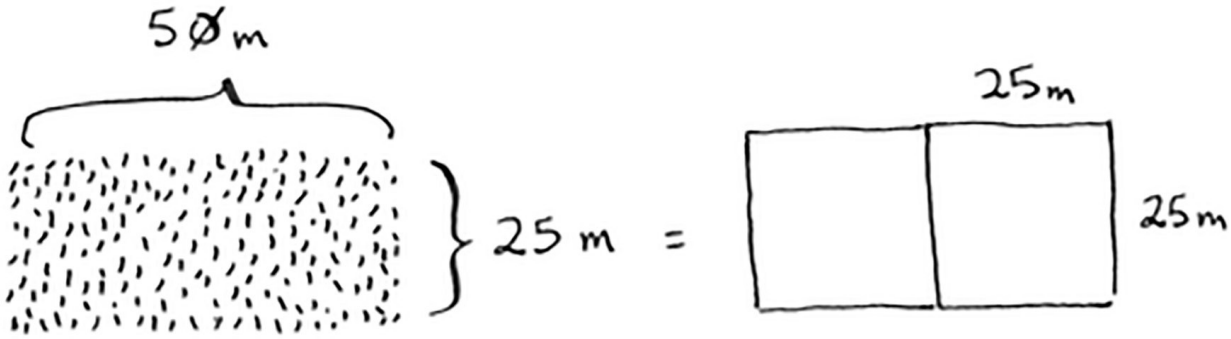
3 . D&C can take some time to grasp. So, we'll do three examples. First I'll show you a visual example. Then I'll do a code example that is less pretty but maybe easier. Finally, we'll go through quicksort, a sorting algorithm that uses D&C.

Suppose you're a farmer with a plot of land.

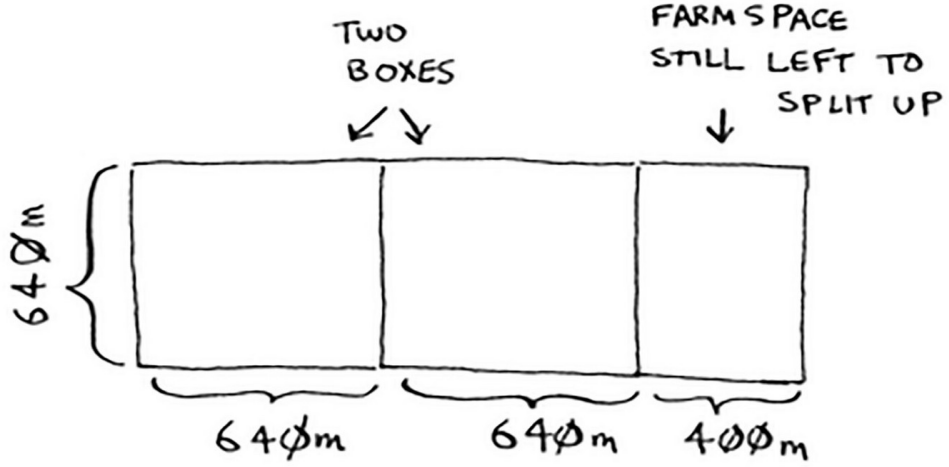


How do you figure out the largest square size you can use for a plot of land? Use the D&C strategy! D&C algorithms are recursive algorithms. To solve a problem using D&C, there are two steps:
1. Figure out the base case. This should be the simplest possible case.
2. Divide or decrease your problem until it becomes the base case.
Let's use D&C to find the solution to this problem. What is the largest square size you can use?

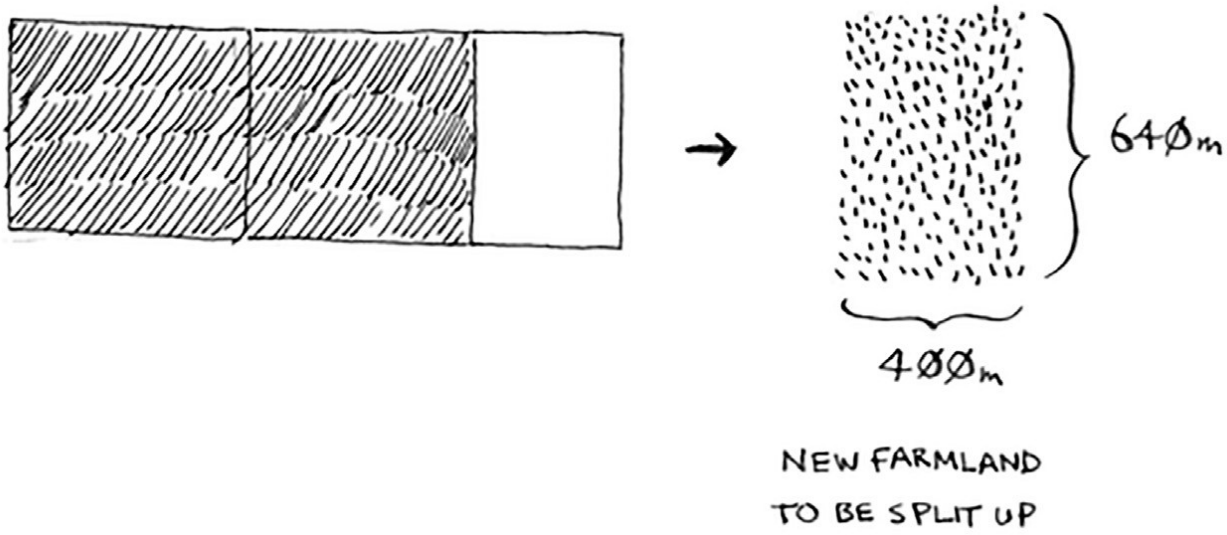
First, figure out the base case. The easiest case would be if one side was a multiple of the other side.
Suppose one side is 25 meters (m) and the other side is 50 m. Then the largest box you can use is 25 m × 25 m. You need two of those boxes to divide up the land.
Now you need to figure out the recursive case. This is where D&C comes in. According to D&C, with every recursive call, you have to reduce your problem. How do you reduce the problem here? Let's start by marking out the biggest boxes you can use.



Suppose one side is 25 meters (m) and the other side is 50 m. Then the largest box you can use is 25 m × 25 m. You need two of those boxes to divide up the land.
Now you need to figure out the recursive case. This is where D&C comes in. According to D&C, with every recursive call, you have to reduce your problem. How do you reduce the problem here? Let's start by marking out the biggest boxes you can use.

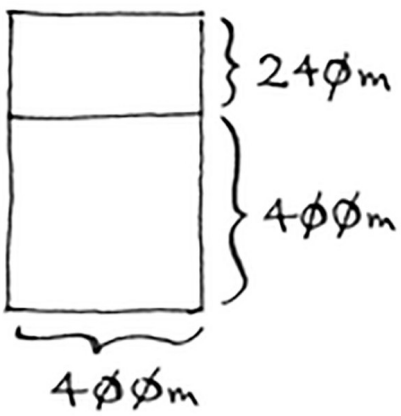


You can fit two 640 × 640 boxes in there, and there's some land still left to be divided. Now here comes the "Aha!" moment. There's a farm segment left to divide. Why don't you apply the same algorithm to this segment?
So you started out with a 1680 × 640 farm that needed to be split up. But now you need to split up a smaller segment, 640 × 400. If you find the biggest box that will work for this size, that will be the biggest box that will work for the entire farm. You just reduced the problem from a 1680 × 640 farm to a 640 × 400 farm!

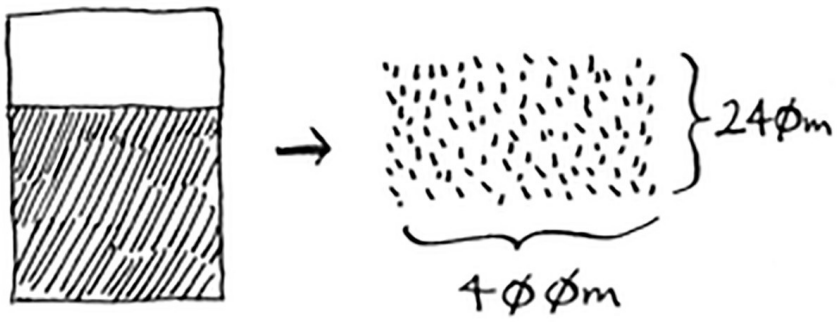


So you started out with a 1680 × 640 farm that needed to be split up. But now you need to split up a smaller segment, 640 × 400. If you find the biggest box that will work for this size, that will be the biggest box that will work for the entire farm. You just reduced the problem from a 1680 × 640 farm to a 640 × 400 farm!

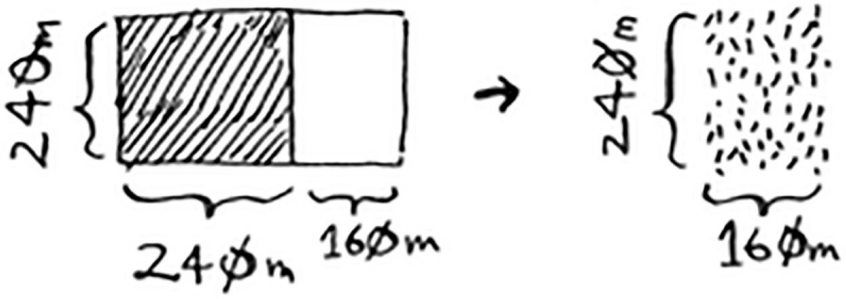
Let's apply the same algorithm again. Starting with a 640 × 400m farm, the biggest box you can create is 400 × 400 m.



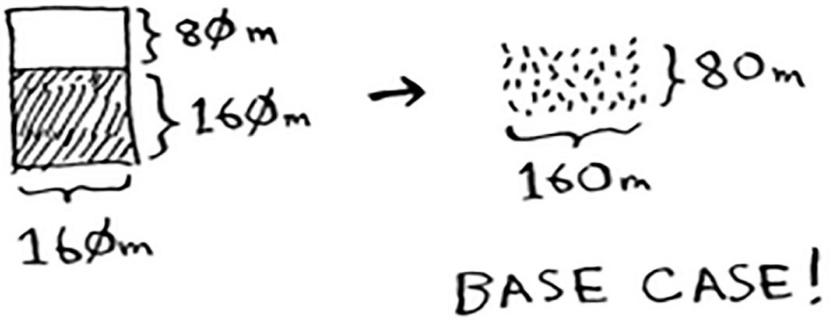
And that leaves you with a smaller segment, 400 × 240 m.



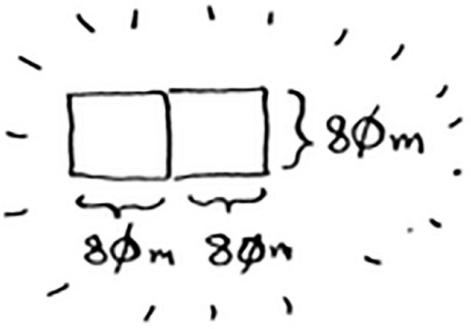
And you can draw a box on that to get an even smaller segment, 240 × 160 m.



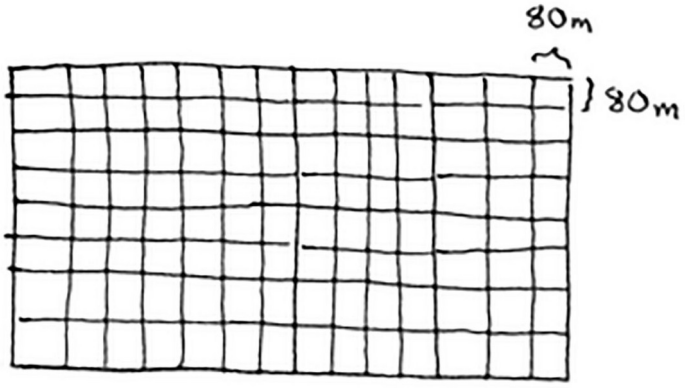
And then you draw a box on that to get an even smaller segment.



Hey, you’re at the base case: 80 is a factor of 160. If you split up this segment using boxes, you don’t have anything left over!

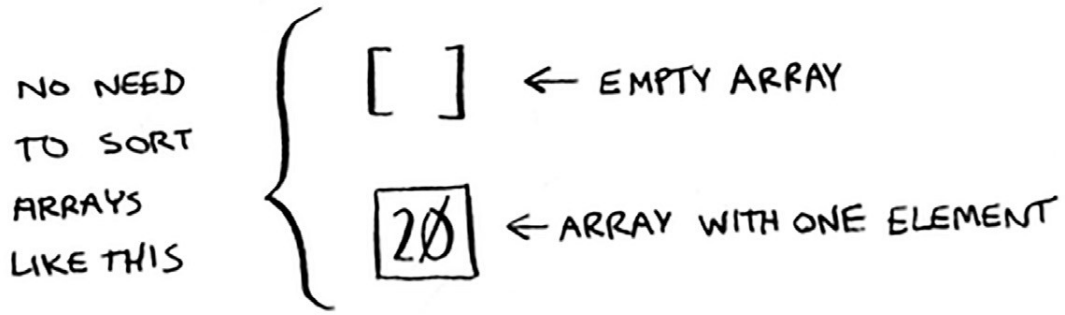


So, for the original farm, the biggest plot size you can use is 80 × 80 m.



part2: Quicksort

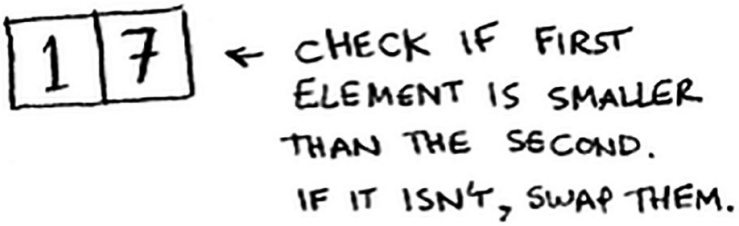
Quick sort is a sorting algorithm. It's much faster than selection sort and is frequently used in real life. For example, the C standard library has a function called qsort , which is its implementation of quicksort. Quicksort also uses D&C.
Let's use quicksort to sort an array. What's the simplest array that a sorting algorithm can handle (remember my tip from the previous section)? Well, some arrays don't need to be sorted at all.



Empty arrays and arrays with just one element will be the base case. You can just return those arrays as is—there's nothing to sort:

```
def quicksort(array):
if len(array) < 2:
return array
```

Let's look at bigger arrays. An array with two elements is pretty easy to sort, too.



What about an array of three elements?

Remember, you're using D&C. So you want to break down this array until you're at the base case. Here's how quicksort works. First, pick an element from the array. This element is called the pivot.

We'll talk about how to pick a good pivot later. For now, let's say the first item in the array is the pivot.

Now find the elements smaller than the pivot and the elements larger than the pivot.
[15,10] [33] []

This is called partitioning. Now you have

• A sub-array of all the numbers less than the pivot

• The pivot

• A sub-array of all the numbers greater than the pivot

The two sub-arrays aren't sorted. They're just partitioned. But if they were sorted, then sorting the whole array would be pretty easy.

If the sub-arrays are sorted, then you can combine the whole thing like this— left array + pivot + right array —and you get a sorted array. In this case, it's [10, 15] + [33] + [] =[10, 15, 33] , which is a sorted array.

How do you sort the sub-arrays? Well, the quicksort base case already knows how to sort arrays of two elements (the left sub-array) and empty arrays (the right sub-array). So if you call quicksort on the two sub-arrays and then combine the results, you get a sorted array!

Quicksort ([15, 10]) + [33] + quicksort([]) > [10, 15, 33] --->A sorted array

This will work with any pivot. Suppose you choose 15 as the pivot instead.
[10][**15**][33]

Both sub-arrays have only one element, and you know how to sort those. So now you know how to sort an array of three elements. Here are the steps:
1. Pick a pivot.
2. Partition the array into two sub-arrays: elements less than the pivot and elements greater than the pivot.
3. Call quicksort recursively on the two sub-arrays.
What about an array of four elements?

[33,10,15,7]

Suppose you choose 33 as the pivot again.
The array on the left has three elements. You already know how to sort an array of three elements: call quicksort on it recursively.

[10,15,7] [33] []

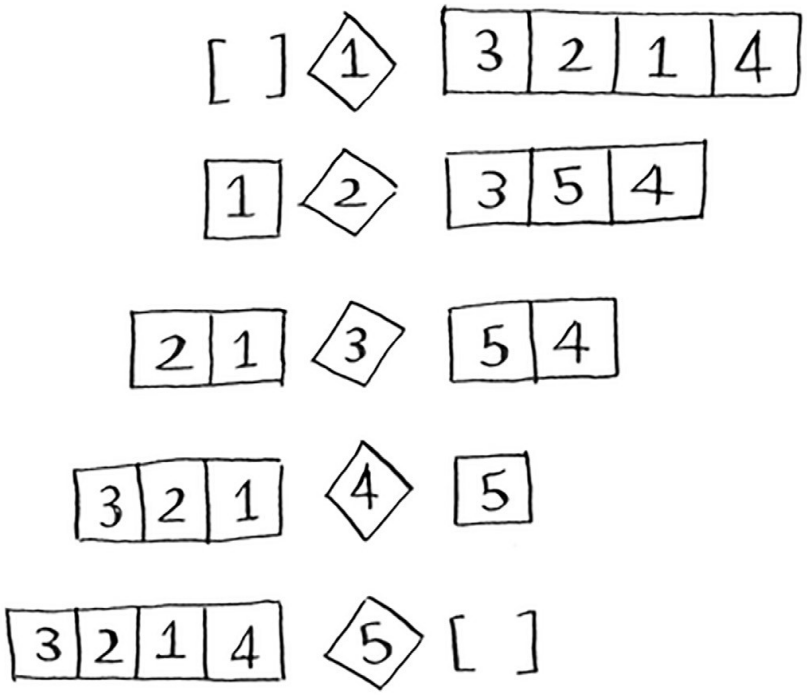
The array on the left has three elements. You already know how to sort an array of three elements: call quicksort on it recursively.

[10,15,7] [33] []
[7][10][15] [33] []

So you can sort an array of four elements. And if you can sort an array of four elements, you can sort an array of five elements. Why is that? Suppose you have this array of five elements.

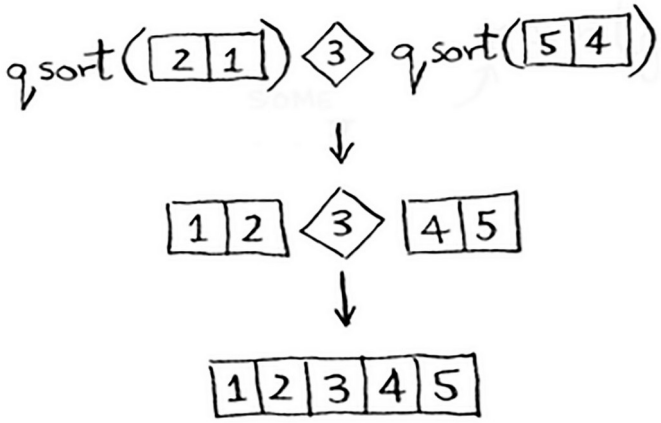
[3,5,2,1,4]

Here are all the ways you can partition this array, depending on what pivot you choose.
Notice that all of these sub-arrays have somewhere between 0 and 4 elements. And you already know how to sort an array of 0 to 4 elements using quicksort! So no matter what pivot you pick, you can call quicksort recursively on the two sub-arrays

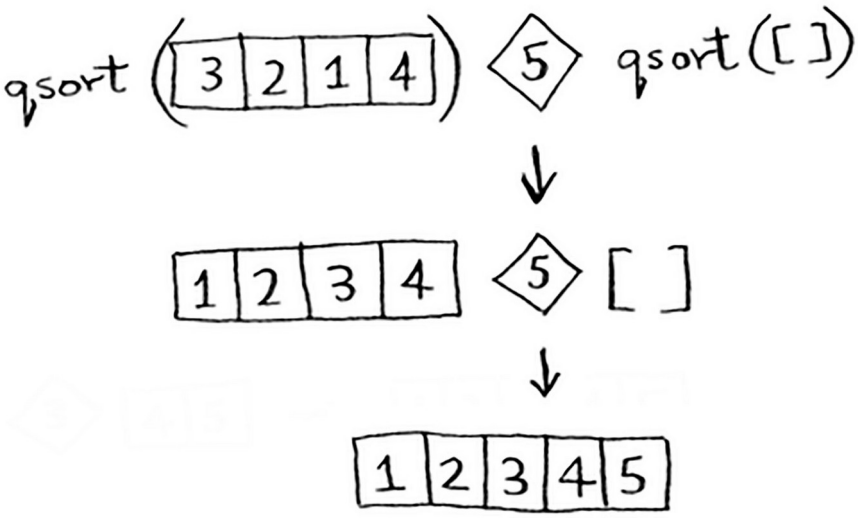


Notice that all of these sub-arrays have somewhere between 0 and 4 elements. And you already know how to sort an array of 0 to 4 elements using quicksort! So no matter what pivot you pick, you can call quicksort recursively on the two sub-arrays.

For example, suppose you pick 3 as the pivot. You call quicksort on the sub-arrays.



The sub-arrays get sorted, and then you combine the whole thing to get a sorted array. This works even if you choose 5 as the pivot.



This works with any element as the pivot. So you can sort an array of five elements. Using the same logic, you can sort an array of six elements, and so on.

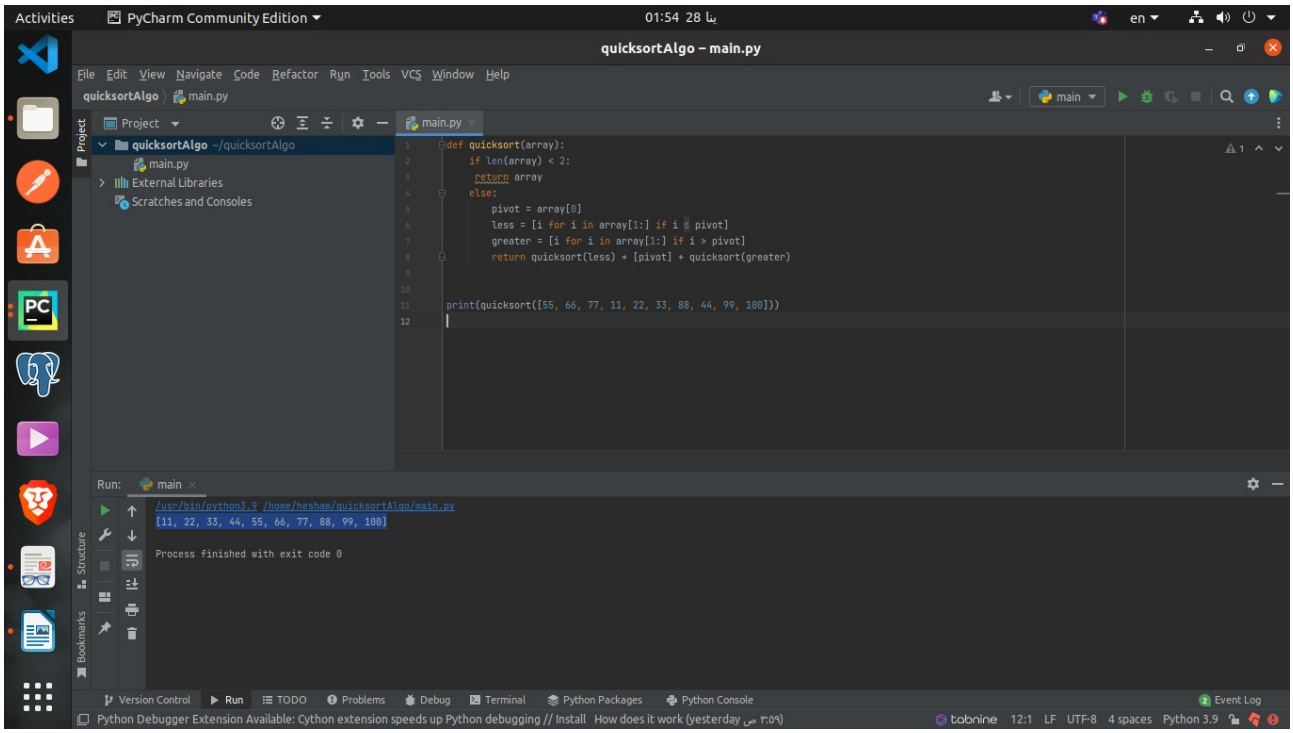
Here's the code for quicksort:

```
def quicksort(array):
    if len(array) < 2:
     return array
    else:
        pivot = array[0]
        less = [i for i in array[1:] if i <= pivot]
        greater = [i for i in array[1:] if i > pivot]
        return quicksort(less) + [pivot] + quicksort(greater)


print(quicksort([55, 66, 77, 11, 22, 33, 88, 44, 99, 100]))
```

output:
[11, 22, 33, 44, 55, 66, 77, 88, 99, 100]

screenshot:


Activities
PyCharm Community Edition
01:54 28 يناير
en
quicksortAlgo – main.py
File Edit View Navigate Code Refactor Run Tools VCS Window Help
quicksortAlgo main.py
main
Project
quicksortAlgo ~/quicksortAlgo
main.py
External Libraries
Scratches and Consoles
main.py
def quicksort(array):
if len(array) < 2:
return array
else:
pivot = array[0]
less = [i for i in array[1:] if i <= pivot]
greater = [i for i in array[1:] if i > pivot]
return quicksort(less) + [pivot] + quicksort(greater)
print(quicksort([55, 66, 77, 11, 22, 33, 88, 44, 99, 100]))
Run: main
/usr/bin/python3.9 /home/hesham/quicksortAlgo/main.py
[11, 22, 33, 44, 55, 66, 77, 88, 99, 100]
Process finished with exit code 0
Version Control
Run
TODO
Problems
Debug
Terminal
Python Packages
Python Console
Event Log
Python Debugger Extension Available: Cython extension speeds up Python debugging // Install How does it work (yesterday ص ٣:٠٩)
tabnine
12:1
LF
UTF-8
4 spaces
Python 3.9