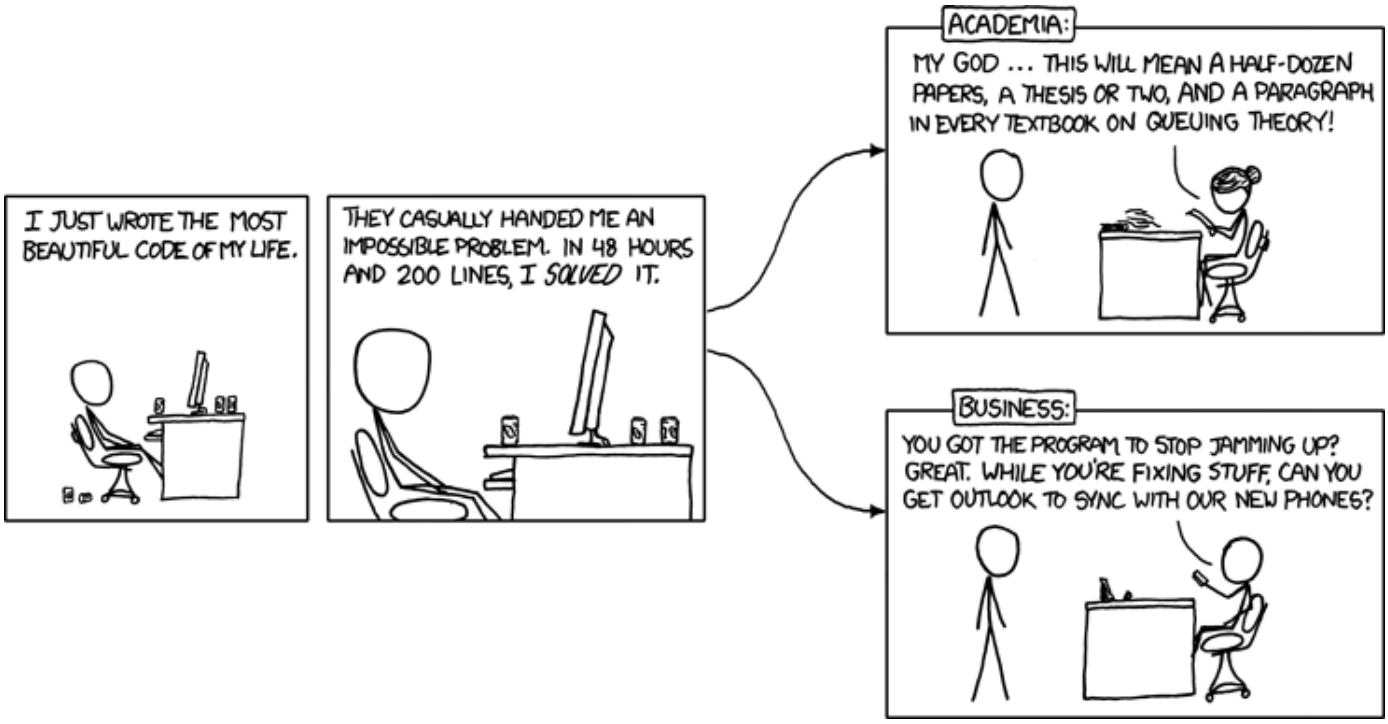
Sorting Unordered Lists: Quick Sort

With a name like that, we can expect a sorting algorithm that gets half-way around the world while bubble sort is still getting its boots on. It'll take something a bit special to beat Shell sort, though.

A song of sixpence

Necessity is the mother of invention, and just like the fast inverse square-root of Quake II, quick sort was born not in a university but out in the wild, by a jobbing programmer trying to solve a problem. Tony Hoare was frustrated by the inefficiencies of shuttle sort for ordering Russian words in his translation program while working in the Soviet Union in 1959, and came up with quick sort to make his code work better. Shortly afterwards, when his boss tasked him with implementing the recently published Shell sort for another project, he casually mentioned that he could do better, and his boss bet him sixpence that he couldn't. 60 years later, we're still reaping the benefits of that 6p investment.



"Some engineer out there has solved P=NP and it's locked up in an electric eggbeater calibration routine. For every 0x5f375a86 we learn about, there are thousands we never see." *XKCD*

The big idea

Sorting algorithms perform fine on short lists, but their performance drops significantly as the length of list increases. A list with twice the length usually costs more than twice as much in CPU time. Reminds me of the sort of nonsense special offer you occasionally see:

If you wanted two such products, you could side-step this dubious special offer by simply purchasing them one at a time. That's essentially what quick sort aims to do. Rather than sort a 1000-item list, why not sort two 500-item lists? Or, better yet, ten 100-item lists? The only catch is ensuring that the first 100 items all belong at the beginning of the list...

The pivot

Quick sort uses the idea of a *pivot* element: a single element in the list (usually the first) is chosen to be the pivot, and every other element is compared with that pivot element, moving it to one of two sub-lists; one for elements smaller than the pivot, one for elements larger (or equal). Having carried out this process, we know a couple of things: Firstly, the pivot element, sandwiched between the two sub-lists, is now in the right position, and we can safely leave it alone for the remainder of the process (we refer to it rather poetically as a 'dead pivot': it has served its purpose, and found its final resting place). Secondly, each of the sub-lists is in the correct position relative to the pivot and relative to the other sub-list. All that remains is to sort each individual sub-list somehow.

Recursion

One of the elegant principles behind quick sort (and, in fact, behind a lot of very efficient algorithms) is that of *recursion*. The process described above breaks one list into two, plus a pivot in between. If we can sort those two lists in some way, we can just reconnect them, with the pivot in between, to get a fully sorted list. But how should we sort those sub-lists? Donald Shell resorted to a shuttle sort for his sub-lists, but Tony Hoare defined his algorithm *in terms of his own algorithm*. How should we sort those pesky sub-lists? With quick sort, of course.

When will it all end?

A classic beginner's error when writing a recursive program is to forget how it will end. Consider this elegant approach to calculating factorials:

```
def factorial(n):
  return n * factorial(n - 1)
```

It's beautiful in its simplicity. If you need to calculate $5!$, just work out $5 \times 4!$. And to find $4!$, we can just do $4 \times 3!$, and so on.

There's a slight snag, though – this program never ends:

```
def factorial(n):
  print(f"Finding {n}! by invoking {n-1}!")
  return n * factorial(n - 1)

print(factorial(5))
```

```
Finding 5! by invoking 4!
Finding 4! by invoking 3!
Finding 3! by invoking 2!
Finding 2! by invoking 1!
Finding 1! by invoking 0!
Finding 0! by invoking -1!
Finding -1! by invoking -2!
```

This program blissfully continues to call itself like a child who has discovered the joy of repeating the word 'poo' incessantly until Python eventually steps in like the parent who finally runs out of patience and puts a stop to this inappropriate behaviour:

```
Finding -989! by invoking -990!
Finding -990! by invoking -991!
Traceback (most recent call last):
  File "main.py", line 7, in <module>
    factorial(5)
  File "main.py", line 5, in factorial
    return n * factorial(n - 1)
  File "main.py", line 5, in factorial
    return n * factorial(n - 1)
  File "main.py", line 5, in factorial
    return n * factorial(n - 1)
  [Previous line repeated 992 more times]
  File "main.py", line 4, in factorial
    print(f"Finding {n}! by invoking {n-1}!")
RecursionError: maximum recursion depth exceeded
while calling a Python object
```

All about that base

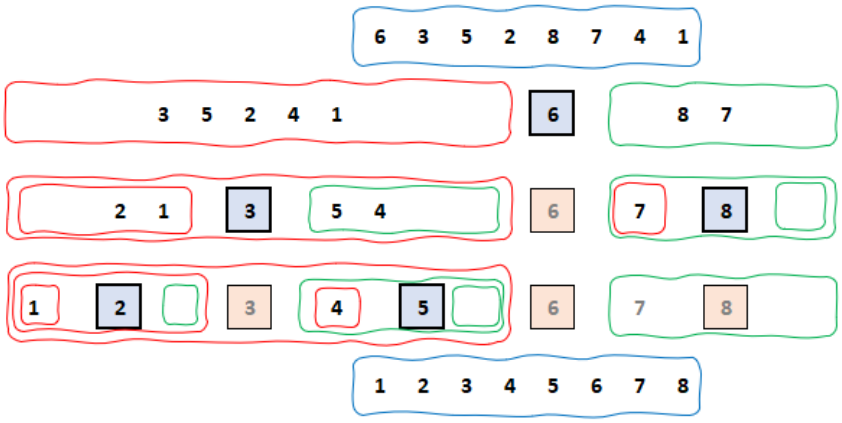
The trick is to consider the 'base case': the end of the line which recursion finally reaches:

In the case of our **factorial** function, we can use the fact that **factorial(0)** is defined to be equal to 1.

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

For quick sort, this base case (or 'terminating case') would be a sub-list which has either 0 or 1 elements. In these cases, the list is clearly already sorted.

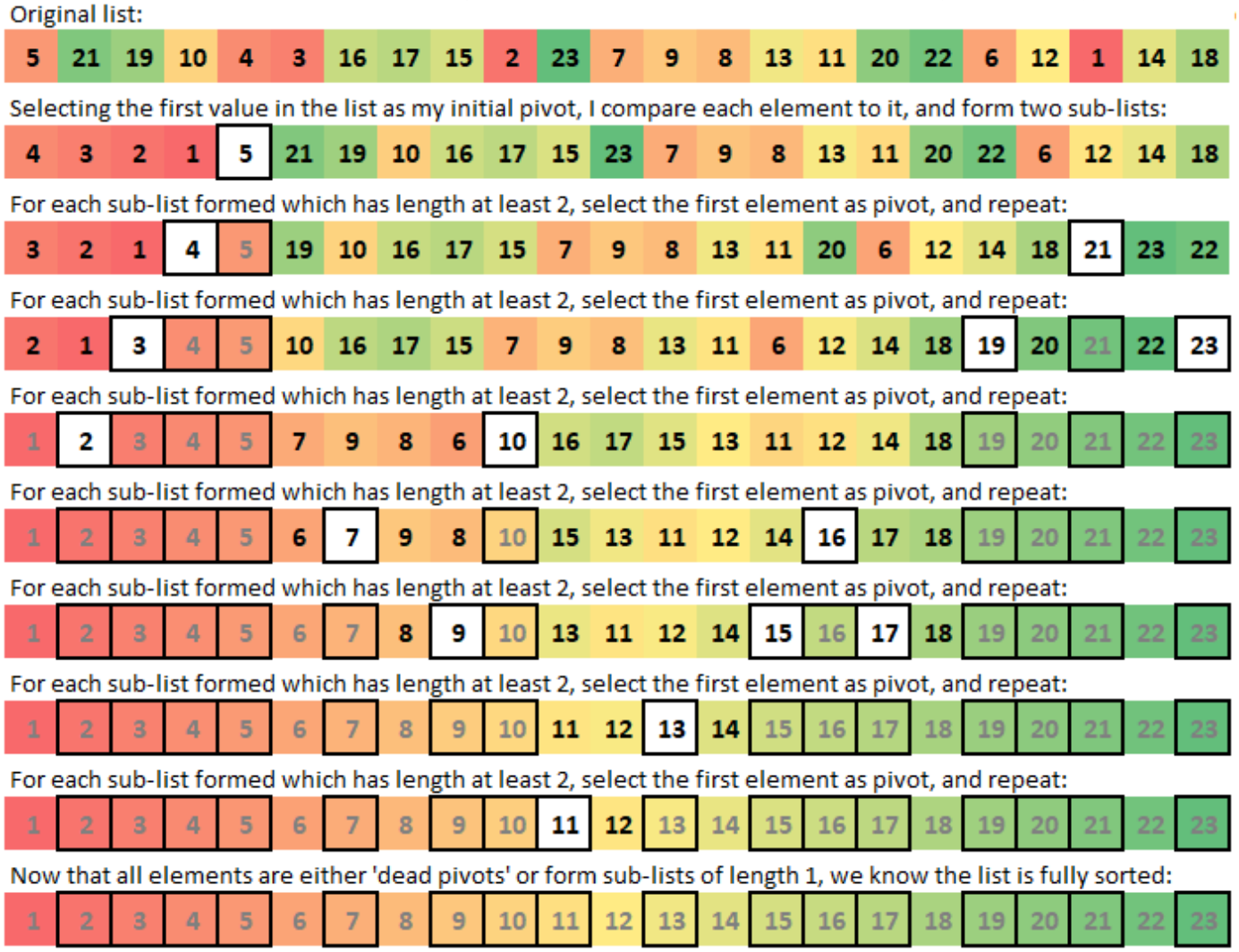
A quick example



The first element, 6, is selected to be the first pivot.
All values below 6 form a sub-list on the left, and values above form their own sub-list on the right. The pivot is now in its proper place ('dead').
Empty or single-item sub-lists are ignored.

Another, longer, example

This example uses a list of length 23 (the same list used for our Shell sort example). For each list, a pivot is chosen, sub-lists created, then the process repeated until every sub-list is trivial. Performance for this length of list is similar to Shell (92 comps, 56 swaps).



Building the full algorithm with Python

First things first

We often begin by thinking about the components that need to be in place before we can write our final program, but in the case of recursion, the function we need is the function we are writing, so although it may feel strange, often the best way with a recursive function is just to dive straight in: let's make a function that:

- Selects a pivot (the first value in the list).
- Compares each other element to the pivot, moving the smaller to a new sub-list.
- Invokes itself on each sub-list to replace them with sorted versions of themselves.
- Puts the two sub-lists back together, with the original pivot in between
- Returns the finished product: a fully sorted list.

But don't forget to include the base case, right at the start: imagine the list provided is so small that it contains either 0 or 1 element:

- If the list is shorter than 2 elements, do nothing, and simply return the whole list.

```
1   def quick(l):
2     if len(l) < 2:
3       return l
4     else:
5       pivot = l[0]
6       above = l[1:]
7       below = []
8       for e in above.copy():
9         if e < pivot:
10          above.remove(e)
11          below.append(e)
12      return quick(below) + [pivot] + quick(above)
```

Lines 2 and 3 cover the 'base case'. Line 5 identifies the pivot, and line 6 creates a new list from the remaining (non-pivot) items. Line 7 creates an empty sub-list for values less than the pivot. Line 8 uses a *copy* of **above** so that the **for** loop doesn't give us problems when we start removing items from **above** on line 10.

The **if** block starting at line 9 compares each element to the pivot, and, if necessary, removes it from the **above** sub-list to the **below** sub-list. The final line is where the recursive part comes in: we put together the three elements: a *sorted* version of **below**, followed by the **pivot**, followed by a *sorted* version of **above**. Notice that lists can be added to produce longer lists (which is why we used **[pivot]**, a list containing the single element **pivot**: we can't add lists to non-lists).

Sorting in-place or not

The built-in Python sorting methods recognize that sometimes we need a list to be sorted 'in place' (that is, the original list is replaced with a sorted version of itself), and sometimes we want to preserve the original list, but generate a new sorted version in addition:

sorted is a built-in function that returns a sorted version of your list, without modifying the original stored list:

```
a = [4, 3, 1, 2]
print(sorted(a)) # shows [1, 2, 3, 4]
print(a)         # shows [4, 3, 1, 2]
```

sort is a built-in list method that acts on the given list, overwriting it with a sorted version of itself:

```
a = [4, 3, 1, 2]
a.sort()         # list overwritten
print(a)         # shows [1, 2, 3, 4]
```

We have been designing our programs in such a way that they *don't* modify the original list. This allows for more flexibility, since the user can choose whether to overwrite their list with our sorted version or not. We have used **l.copy()** previously, but slicing (as in line 6, where **above** is formed from a slice of the list **l**) also works.

Speak to me!

It's all very well having a program that sorts numbers, and we can see if there are any glaring errors pretty quickly by looking at the resulting list, but if we want to check the nuts and bolts, it would be nice to have the code equivalent of one of those plug-in diagnostic tools the garage use when your car's check-engine light comes on. Fortunately, we have the **print** function for that. We can use it to make a 'with detail' version of our quick sort function:

```
1   def quick(l):
2     print(f"Running quick({l}):")
3     if len(l) < 2:
4       print(f">Base case: returning same list.")
5       return l
6     else:
7       pivot = l[0]
8       above = l[1:]
9       below = []
10      print(f" Pivot: {pivot}")
11      for e in above.copy():
12        print(f"  Comparing {e} to {pivot}...")
13        if e < pivot:
14          above.remove(e)
15          below.append(e)
16          print(f"   Moving {e} to 'below'.")
17      print(f">Returning sorted list")
18      return quick(below) + [pivot] + quick
        (above)
19
20  print(quick([3, 2, 4, 1]))
```

```
Running quick([3, 2, 4, 1]):
 Pivot: 3
  Comparing 2 to 3...
   Moving 2 to 'below'.
  Comparing 4 to 3...
  Comparing 1 to 3...
   Moving 1 to 'below'.
>Returning sorted list
Running quick([2, 1]):
 Pivot: 2
  Comparing 1 to 2...
   Moving 1 to 'below'.
>Returning sorted list
Running quick([1]):
>Base case: returning same list.
Running quick([]):
>Base case: returning same list.
Running quick([4]):
>Base case: returning same list.
[1, 2, 3, 4]
>
```

Comps and Swaps

A running commentary using **print** can be really helpful for debugging and optimising, and just getting a sense of what is going on behind the scenes, but of course it gets somewhat unwieldy for lists longer than a handful of items. What we could do with is a summary of the important stats: comparisons and swaps. Have a go at adapting your code so that it keeps track of these, and returns them at the end along with the list.

```
1   def quick(l):
2     c, s = 0, 0
3     if len(l) < 2:
4       return l, c, s
5     else:
6       pivot = l[0]
7       above = l[1:]
8       below = []
9       for e in above.copy():
10        c += 1
11        if e < pivot:
12          s += 1
13          above.remove(e)
14          below.append(e)
15      new_below, b_c, b_s = quick(below)
16      new_above, a_c, a_s = quick(above)
17      c += (b_c + a_c)
18      s += (b_s + a_s)
19      return new_below + [pivot] + new_above, c, s
```

We've made a few changes:
At the very start, we initialise **c** and **s** to keep track of comparisons and swaps.
Both **return** statements have been modified to give not just the sorted list, but a three item tuple: the sorted list, **c** and **s**.

And finally, since we need to keep track of comps and swaps all the way up the chain, we extract that information at the recursion steps and include it in the final reported statistics.

What's the score?

As we did with the previous algorithms, we are now going to bolt on some extra functionality to allow a quick performance analysis for random lists of varying sizes.

```
def random_test(num_lists, length):
    comparisons, swaps = [], []
    for i in range(num_lists):
        l = [random.randint(1, 100) for i in range(length)]
        _, c, s = quick(l)
        comparisons.append(c)
        swaps.append(s)
    return comparisons, swaps

c, s = random_test(10, 23)
print(f"Comparisons: {c} (average: {sum(c) / len(c)})")
print(f"Swaps: {s} (average: {sum(s) / len(s)})")
```

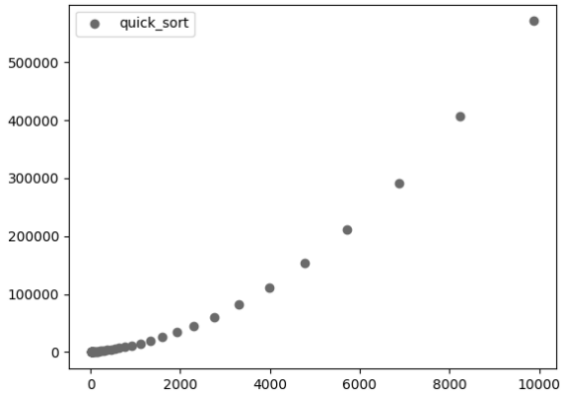
This code should look very familiar. I've basically taken the same code used for Shell sort, but changed the name of the function being used to sort the lists. Another advantage of keeping a consistent format and structure.

*Don't forget to import the **random** module!*
*You'll also need **matplotlib.pyplot** for the next bit.*

```
import random
import matplotlib.pyplot as plt
```

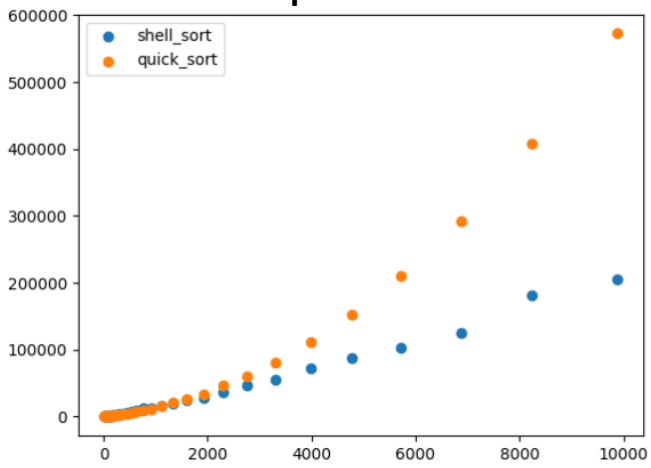
We can also re-use the code from Shell sort to see how quick sort scales with list length:

```
33  lengths, comps = [], []
34  length = 10
35  while length <= 10000:
36      c, _ = random_test(10, length)
37      print(f"For a random list of length {length}:")
38      print(f"Average comps: {sum(c) / len(c)}")
39      lengths.append(length)
40      comps.append(sum(c) / len(c))
41      length = int(length * 1.2)
42
43  plt.scatter(lengths, comps)
44  plt.savefig("Comparisons vs Lengths Quick sort.png")
```

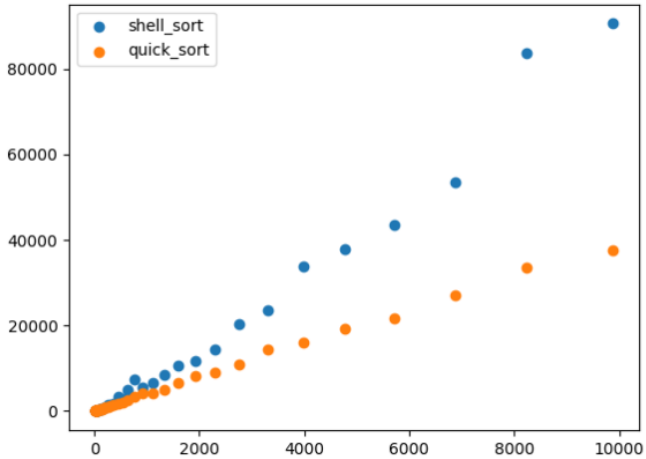


That doesn't look as good as expected… let's bring in the Shell sort results for comparison:

Comparisons:

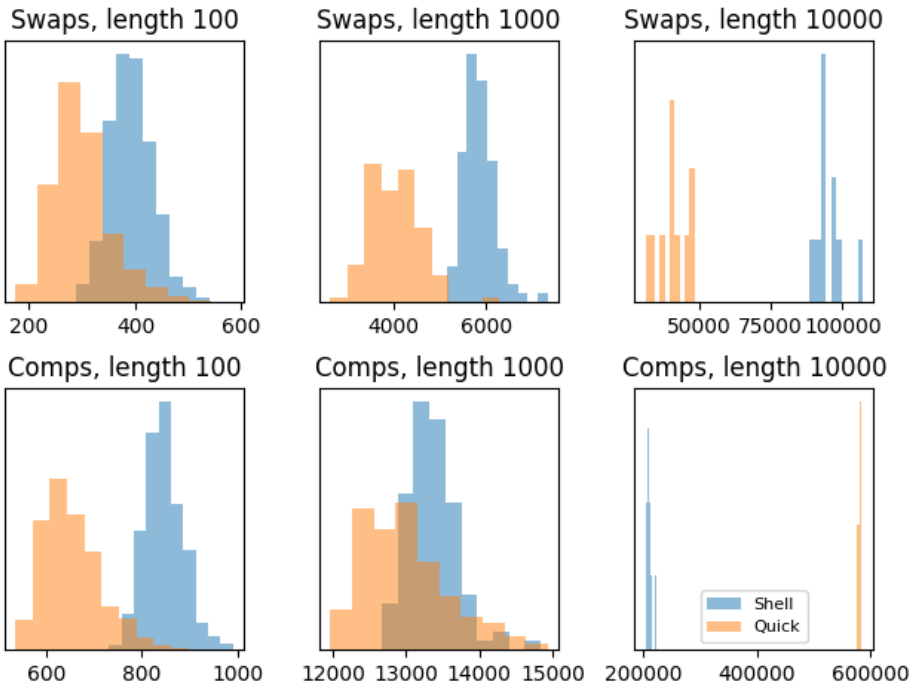


Swaps:



As far as comparisons go, quick sort isn't really living up to its name, but it certainly has Shell beat when it comes to swaps. Don't forget, however, that, in general, swaps are more expensive for a computer than comparisons. It will vary from machine to machine, but my calculations on repl.it make it a 2: 1 ratio, giving quick sort the edge by a factor of 15% for 1000 items, but more than 50% slower for 10,000 items. What's going wrong?

We can compare Shell and quick sort for a range of list lengths:



But this isn't very promising. I wonder what could be going wrong…

Tony won sixpence with this algorithm, but based on this analysis it's not clear that quick sort should win the prize. As we saw with the previous graphs, while it beats Shell on swaps, it loses big-time on comparisons.

Duplicate elements

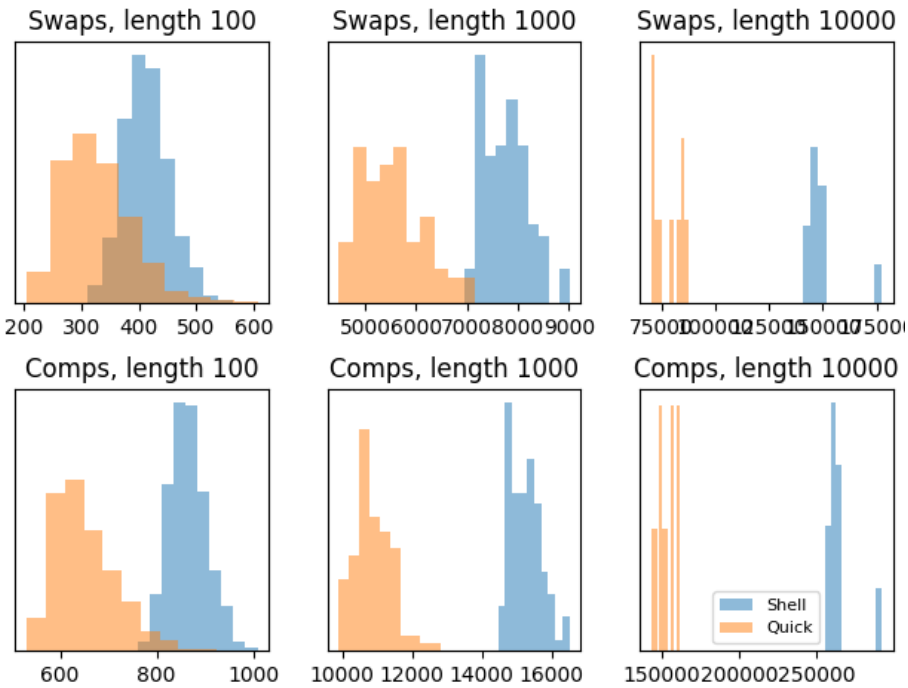
One of the weaknesses of quick sort is revealed when sorting a list with multiple duplicate elements. We have inadvertently done this by populating our lists with random integers between 1 and 100. For a list of length 100, that's not a problem, and it's still not much of a problem for lists of length 1000: there will be around 10 of each integer. But for 10,000 items? Now we have about a hundred 1s, a hundred 2s, etc, and quick sort isn't so quick. The easiest way around this, for a fairer comparison, is to change that line in **random_test**:

Replace this…

```
def random_test(num_lists, length):
    comparisons, swaps = [], []
    for i in range(num_lists):
        l = [random.randint(1, 100) for i in range(length)]
        _, c, s = quick(l)
        comparisons.append(c)
        swaps.append(s)
    return comparisons, swaps
```

… with this

```
def random_test(num_lists, length):
    comparisons, swaps = [], []
    for i in range(num_lists):
        l = [random.random() for i in range(length)]
        _, c, s = quick(l)
        comparisons.append(c)
        swaps.append(s)
    return comparisons, swaps
```



That's more like it! Both swaps and comparisons show significant improvement compared to Shell. Looks like Tony deserved his sixpence after all. Note that horizontal scales do not start from zero, so the improvement has been somewhat exaggerated here, but quick is about 25% cheaper for 100-item lists, and 50% cheaper overall for 10,000-item lists.

Additional info and how to write out an implementation of the Quick Sort algorithm:

Name:	Quick sort, like shell sort, is a 'divide-and-conquer' algorithm, designed to be really efficient. Also known as the 'partition-exchange' sort since each pivot partitions the remaining items.
Summary:	Choose a pivot and compare each item to it, forming two sub-lists. Recursively apply the algorithm to each sub-list until all items have been pivots and are therefore in place.
Efficiency:	On average: $O(n \log n)$ For worst case: $O(n^2)$ (but this is fairly rare) Although traditionally (and in our implementation) the first element of any sub-list is chosen as a pivot, in mostly sorted or reverse order lists this results in worst case behaviour, so the median of the first, middle and last values is often chosen instead. Another common optimisation is to use shuttle sort for sub-lists that are sufficiently small, since it is a very cheap algorithm for small lists.
Algorithm:	Set the first item as a pivot and (without reordering) compare all subsequent numbers in the list with the pivot, adding the lower to a left-most list and the higher to a right-most list. Choose a pivot for each sub-list and repeat the procedure on each sub-list until all sub-lists contain only one element. Each pivot will end up in the correct place after use.
Example:	Sort the list 8 3 2 6 9 4 2 7 using quick sort. `Pivots:` **8** 3 2 6 9 4 2 7 `First pass:` 3 2 6 4 2 7 **8** 9 (comparisons: 7, swaps: 6) *Note: you could underline current pivots & circle dead ones* `Pivots:` **3** 2 6 4 2 7 **8** 9 `Second pass:` 2 2 **3** 6 4 7 **8** 9 (comparisons: 5, swaps: 2) `Pivots:` **2** 2 **3** **6** 4 7 **8** 9 `Third pass:` **2** 2 **3** 4 **6** 7 **8** 9 (comparisons: 3, swaps: 1) **Totals: comparisons: 15, swaps: 9** Note: Sublists of 1 can be ignored since they are automatically in the correct place.
Visual:	For more details and an animation of this algorithm for a number of different cases, see: www.sorting-algorithms.com