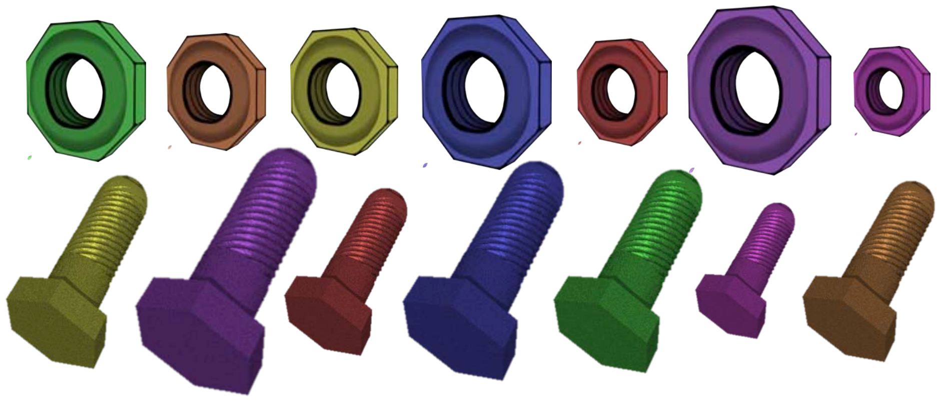
9. The Disorganized Handyman

A bad handyman always blames his tools. – Famous Proverb.
What if my hammer is made of paper? Can I blame it then? – Author Unknown.

Programming constructs and algorithmic paradigms covered in this puzzle: In-place pivoting. Recursive in-place sorting.

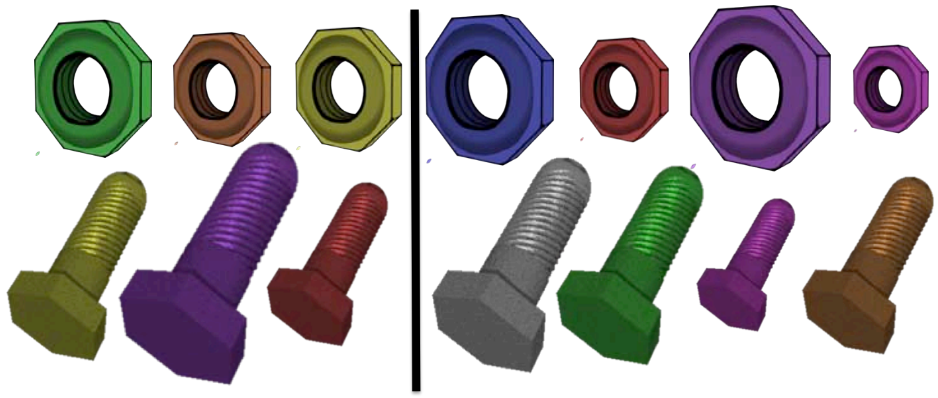
A handyman has a whole collection of nuts and bolts of different sizes in a bag. Each nut is unique and has a corresponding unique bolt, but the disorganized handyman has dumped them all into one bag and they are all mixed up. How best to "sort" these nuts and attach each to its corresponding bolt?



Given n nuts and n bolts, the handyman can pick an arbitrary nut and try it with each bolt and find the one that fits the nut. Then, he can put away the nut-bolt pair, and he has a problem of size n – 1. This means that he has done n "comparisons" to reduce the problem size by 1. n – 1 comparisons will then shrink the problem size to n – 2, and so on. The total number of comparisons required is n + (n – 1) + (n – 2) + … + 1 = n(n+1)/2. You could argue the last comparison is not required since there will only be one nut and one bolt left, but we will call it a confirmation comparison.

Can one do better in terms of number of comparisons required? More concretely, can one split the nuts and bolts up into two sets, each of half the size, so we have two problems of size n/2 to work on? This way, if the handyman has a helper, they can work in parallel. Of course, we could apply this strategy recursively to each of the problems of size n/2 if there are additional kind people willing to help.

Unfortunately, simply splitting the nuts up into two (roughly) equal sized piles A and B and the bolts into similar sized piles C and D does not work. If we group nuts and bolts corresponding to A and C together into a nut-bolt pile, it is quite possible that a nut in A may not fit *any* bolt in C; the correct bolt for that nut is in D. Here's an example.



Think of the A, C piles as the ones to the left and the B, D piles as the ones to the right. The biggest bolt is in the left pile (second from left) and the biggest nut is in the right pile (second from right) ☹

Can you think of a recursive Divide and Conquer strategy to solve the Nuts and Bolts problem so you require significantly fewer than n(n+1)/2 *comparisons when* n *is large?*

In devising a Divide and Conquer strategy one has to determine how to divide the problem so the subproblems are essentially the same as the original problem, except smaller. In the Nuts and Bolts problem, an arbitrary division of nuts that is unrelated to the division of bolts will not work. We have to somehow guarantee that the subproblems can be solved independently of each other and this, of course, means that each bolt that attaches to each nut in the subproblem's nut-bolt collection has to be in the collection.

Pivoting in Divide and Conquer

Pivoting is the key idea that results in a Divide and Conquer algorithm for our problem. What we can do is to pick a bolt – we will call it the *pivot bolt* – and use it to determine which nuts are smaller, which one fits exactly, and which nuts are bigger. We separate the nuts into three piles in this way, with the middle pile being of size 1 and containing the paired nut. Therefore, in this process we have discovered one nut-bolt pairing. Using the paired nut, that we will call the *pivot nut*, we can now split the bolts into two piles, the bolts that are bigger than the pivot nut, and those that are smaller. The bigger bolts are grouped with the nuts that were bigger than the pivot bolt, and the smaller bolts are grouped with the nuts that were smaller than the pivot bolt.

We now have a pile of "big" nuts and "big" bolts, all together, and a pile of small nuts and small bolts all together. Depending on the choice of the pivot bolt, there will be a differing number of nuts in the two piles. However, we are guaranteed that the number of nuts is the same as the number of bolts in each pile if the original problem had a matched set of nuts and bolts, and moreover, the nut corresponding to any bolt in the pile is guaranteed to be in the same pile!

In this strategy, we had to make n comparisons given the pivot bolt to split the nuts into two piles. In the process we discover the pivot nut. We then make n – 1 comparisons to split the bolts up and add them to the nut piles. That is a total of 2n – 1 comparisons. Assuming we chose a pivot nut that was middling in size, we have two problems roughly of size n/2. We can subdivide these two problems of size n/2 using roughly n total comparisons to four problems of size n/4.

The cool thing is that the problem sizes halve at each step, rather than only shrinking by 1. For example, suppose n = 100. The original strategy requires 4950 comparisons. In the new strategy, using 199 comparisons, we get two subproblems each roughly of size 50. Even if we use the original strategy for each of these subproblems, we will only require 1225 comparisons for each one, for a total of 199 + 1225 * 2 = 2649 comparisons. Of course, we can perform recursive Divide and Conquer. In fact, if we can split each problem we encounter roughly in half[1], the number of comparisons in the recursive strategy will grow as $n \log_2 n$ as compared to n^2 in the original strategy.

[1] This is not trivial since we have to pick the nut that looks like roughly half the nuts would be larger and half would be smaller in each pile that we want to split.

Interestingly, this puzzle has a deep relationship with perhaps the most widely used sorting algorithm Quicksort. Quicksort relies on the notion of pivoting that we just described.

Relationship to Sorting

In particular, suppose we have a Python list, which is implemented as an array, with unique elements[2] as shown below:

a	*b*	*c*	*d*	*e*	*f*	*g*	*h*

We wish to sort the list in ascending order. We choose an arbitrary pivot element, say, *g*, but it could just as easily been the last element *h*. Now, we will partition the list into two sublists where the left sublist has elements less than *g*, and the right sublist has elements greater than *g*. The two sublists are not sorted, that is the order of elements less than *g* in the left sublist is random. We can now represent the list as:

Elements less than *g*	*g*	Elements greater than *g*

The beautiful observation is that we can sort the left sublist without affecting the position of *g*, and similarly for the right sublist. Once these two sublists have been sorted, we are done!

Let's look at a possible implementation of the recursive Divide and Conquer quicksort algorithm. We first describe the code for the recursive structure and then the code for the pivoting step.

```
def quicksort(lst, start, end):
    if start < end:
        split = pivotPartition(lst, start, end)
        quicksort(lst, start, split - 1)
        quicksort(lst, split + 1, end)
```

The function `quicksort` takes a Python list corresponding to the array to be sorted and the start and end indices of the list. The list elements go from `lst[start]` to `lst[end]`. We could have assumed that the starting index is 0 and the ending index is the length of the list minus 1, but as you will see, asking for the indices as arguments has the advantage of not requiring a wrapper function like we had to have in the N-queens puzzle and the courtyard tiling puzzle. It is important to note that the procedure modifies the argument list `lst` to be sorted and does not return anything.

If `start` equals `end`, then there is only one element in the list, which does not need sorting – that is our base case and we do not need to modify the list. If there are two or

[2] The sorting algorithm and the code that we will present will work for non-unique elements, but it is easier to describe the algorithm assuming unique elements.

more elements, we have to split the list. The function `pivotPartition` selects a pivot element in the list (*g* in our example above) and modifies the list so the elements less than the pivot element are before the pivot and the ones greater than the pivot are after the pivot element. The *index* of the pivot element is returned. If we have the index, then we can effectively split the list simply by telling the recursive calls what the start and end indices are. This is primarily why we have them as arguments to `quicksort`. Since the element at the `end` index is part of the list and we do not have to touch `lst[split]`, the two recursive calls correspond to `lst[start]` to `lst[split - 1]` (Line 4) and `lst[split + 1]` to `lst[end]` (Line 5).

All that remains is to implement `pivotPartition`, which chooses a pivot between the start and end indices and modifies the argument list between the start and end indices appropriately. Here's a first implementation of `pivotPartition`.

```
 1.   def pivotPartition(lst, start, end):
 2.       pivot = lst[end]
 3.       less, pivotList, more = [], [], []
 4.       for e in lst:
 5.           if e < pivot:
 6.               less.append(e)
 7.           elif e > pivot:
 8.               more.append(e)
 9.           else:
10.              pivotList.append(e)
11.       i = 0
12.       for e in less:
13.           lst[i] = e
14.           i += 1
15.       for e in pivotList:
16.          lst[i] = e
17.          i += 1
18.       for e in more:
19.           lst[i] = e
20.           i += 1
21.       return lst.index(pivot)
```

The first line of the body of the function chooses the pivot element as the last element of the list (Line 2). In our Nuts and Bolts problem, we wish to find a middling size pivot, and here too we would like to choose an element such that about half the other elements are smaller and the other half are bigger. We do not want to search for the best pivot because that may require significant computation. If we assume that the input list is initially randomized, then there is equal probability that any element is a "middle" element. Therefore, we pick the last element as the pivot. Note that the order of the smaller elements and bigger elements in the split lists will still be random since we are not sorting them in `pivotPartition`. We can therefore keep picking the last element as the pivot recursively and we will get split lists that are roughly equal in size. This will mean that on average `quicksort` will only require $n \log_2 n$ comparisons to sort the original list with n elements. In a pathological case it may require n^2 operations – we will explore Quicksort's behavior in the exercises.

We have three lists corresponding to the elements that are smaller than the pivot (`less`), the elements that are equal to the pivot (`pivotList`) and the elements that are bigger than the pivot (`more`). `pivotList` is a list because there may be repeated values in the list and one of these may be chosen as a pivot. These three lists are initialized to be empty on Line 3. In Lines 4-10, scanning the input list `lst` populates the three lists. In Lines 11-20 `lst` is modified to have elements in `less` followed by elements in `pivotList` followed by elements in `more`.

Finally, the index of the pivot element is returned. Note that if there are repeated elements in the list and, in particular, if the pivot element is repeated, the index of the first occurrence of the pivot is returned. This means that the second recursive call in `quicksort` (Line 5) will operate on a (sub)list whose first elements are equal to the pivot. These elements will remain at the front of the (sub)list since the other elements are all greater than the pivot.

In-Place Partitioning

The above implementation does not exploit the main advantage of Quicksort in that the partitioning step, i.e., going from the original list/array to the one with *g*'s location fixed and the two sublists unsorted but satisfying ordering relationships with *g*, can be done *without* requiring additional list/array storage.

The Merge Sort algorithm covered in the N Queens puzzle only requires $n \log_2 n$ comparisons in the worst case. Merge Sort guarantees during splitting that two sublists differ in size by at most one element. The merge step of Merge Sort is where all the work happens. In Quicksort, the pivot-based partitioning or splitting is the workhorse step. The merge step is trivial. Merge Sort requires additional storage of a temporary list during its merge step, whereas Quicksort as we will show below does not.

While the Selection Sort algorithm we coded in the Best Time To Party puzzle also does not have to make a copy of the list to be sorted, it is quite slow since it has two nested loops each of which is run (approximately) n times, where n is the size of the list to be sorted. This means that it requires a number of comparisons that grows as n^2, similar to the naïve Nuts and Bolts pairing algorithm we discussed that requires $n(n+1)/2$ comparisons.

Quicksort is one of the most widely deployed sorting algorithm because, on average, it only requires $n \log_2 n$ comparisons, and does not require additional list storage if `pivotPartition` is implemented cleverly as shown below. Other recursive sorting algorithms that only require $n \log n$ comparisons typically need to allocate additional memory that grows with the size of the list to be sorted.

```
 1.   def pivotPartitionClever(lst, start, end):
 2.       pivot = lst[end]
 3.       bottom = start - 1
 4.       top = end
 5.       done = False
 6.       while not done:
 7.           while not done:
 8.              bottom += 1
 9.               if bottom == top:
10.                   done = True
11.                   break
12.               if lst[bottom] > pivot:
13.                   lst[top] = lst[bottom]
14.                   break
15.           while not done:
16.               top -= 1
17.               if top == bottom:
18.                   done = True
19.                   break
20.               if lst[top] < pivot:
21.                   lst[bottom] = lst[top]
22.                   break
23.       lst[top] = pivot
24.       return top
```

This code is quite different from the first version. The first thing to observe about this code is that it works exclusively on the input list `lst` and does not allocate additional list/array storage to store list elements other than the variable `pivot` that stores one list element. (The list variables `less`, `pivotList` and `more` have disappeared.) Furthermore, only list elements between the start and end indices are modified. This procedure uses *in-place* pivoting – the list elements exchange positions and are not copied from one list to another wholesale as in the first version of the procedure.

It is easiest to understand the procedure with an example. Suppose we want to sort the following list:

```
a = [4, 65, 2, -31, 0, 99, 83, 782, 1]
quicksort(a, 0, len(a) - 1)
```

How exactly is the first pivoting done in-place? The pivot is the last element `1`. When `pivotPartitionClever` is called for the first time, it is called with `start = 0` and `end = 8`. This means that `bottom = -1` and `top = 8`. We enter the outer **`while`** loop and then the first inner **`while`** loop (Line 7). The variable `bottom` is incremented to `0`. We search rightward from the left of the list for an element that is greater than the pivot element `1`. The very first element `a[0] = 4 > 1`. We copy over this element to `a[top]`, which contains the pivot. At this point, we have element `4` duplicated in the list, but no worries, we know what the pivot is, since we stored it in the variable `pivot`. If we printed the list and the variables `bottom` and `top` after the first inner **`while`** loop completes, this is what we would see:

```
[4, 65, 2, -31, 0, 99, 83, 782, 4] bottom = 0 top = 8
```

Now, we enter the second inner **`while`** loop (Line 15). We search moving leftward from the right of the list at `a[7]` (the variable `top` is decremented before the search) for an element that is less than the pivot `1`. We keep decrementing `top` till we see the element `0`, at which point `top = 4`, since `a[4] = 0`. We copy over element `0` to `a[bottom = 0]`. Remember that `a[bottom]` was copied over to `a[8]` prior to this so we are not losing any elements in the list. This produces:

```
[0, 65, 2, -31, 0, 99, 83, 782, 4] bottom = 0 top = 4
```

At this point we have taken one element `4` that is greater than the pivot `1` and put it all the way to the right of the list, and we have taken one element `0` which is less than the pivot `1` and put it all the way to the left of the list.

We now go into the second iteration of the outer **`while`** loop. The first inner **`while`** loop produces:

```
[0, 65, 2, -31, 65, 99, 83, 782, 4] bottom = 1 top = 4
```

From the left we found `65 > 1` and we copied it over to `a[top = 4]`. Next, the second inner **`while`** loop produces:

```
[0, -31, 2, -31, 65, 99, 83, 782, 4] bottom = 1 top = 3
```

We moved leftward from `top = 4` and discovered `-31 < 1` and copied it over to `a[bottom = 1]`.

In the second outer **`while`** loop iteration we moved one element `65` to the right part of the list where beyond `65` all elements are greater than the pivot `1`. And we moved `-31` to the left of the list where all elements to the left of `-31` are less than the pivot `1`.

We begin the third iteration of the outer **`while`** loop. The first inner **`while`** loop produces:

```
[0, -31, 2, 2, 65, 99, 83, 782, 4] bottom = 2 top = 3
```

We discovered `a[bottom = 2] = 2` > 1 and moved it to `a[top = 3]`. The second inner **`while`** loop decrements `top` and sees that it is equal to `bottom` and sets `done` to **`True`**, and we break out of the second inner **`while`** loop. Since `done` is **`True`**, we do not continue the outer **`while`** loop.

We set `a[top = 2] = pivot = 1` (Line 23) and return the index of the pivot `1`, which is `2`. The list `a` now looks like:

```
[0, -31, 1, 2, 65, 99, 83, 782, 4]
```

We have indeed pivoted around the element `1` ☺

Of course, all we have done is split the original list `a` up into two lists that are of size 2 and size 6. We need to recursively sort these sublists. For the first sublist of 2 elements, we will pick `-31` as the pivot and produce `-31, 0`. For the second sublist, we will pick `4` as the pivot and the process continues.

Finally, it is important to note that unlike the procedure `pivotPartition`, `pivotPartitionClever` assumes that the pivot is chosen to be the end of the list. So the assignment `pivot = lst[end]` (Line 2) is crucial to correctness!

Exercises

Exercise 1: Modify `pivotPartitionClever` to count the number of element moves and to return the number of moves, in addition to returning the pivot. There are exactly two places where list elements are moved from one location to another in `pivotPartitionClever`. Add up the moves required in all `pivotPartitionClever` calls and print the total number after sorting is completed. This means that the procedure `quicksort` should count the moves made in its `pivotPartitionClever` call and in the two recursive calls that it makes, and should return this count.

The total number of moves when `quicksort` is run on our example list `a = [4, 65, 2, -31, 0, 99, 83, 782, 1]` is 9. To further verify your implementation, run `quicksort` on a list generated by `L = list(range(100))`, which produces an ascending list of numbers from `0` to `99`, and check that no moves are made.

Exercise 2: The number of moves is not the best indicator of computational complexity since moves are only made when the comparisons on Lines 12 and 20 in `pivotPartitionClever` return **`True`**. Count the number of iterations in both inner **`while`** loops of `pivotPartitionClever` using the same methodology as in Exercise 1. Verify that the total number of iterations for both loops across all recursive calls is 24 for list `a = [4, 65, 2, -31, 0, 99, 83, 782, 1]`.

Generate a "randomly ordered" list of 100 numbers deterministically as follows:

```
R = [0] * 100
R[0] = 29
for i in range(100):
    R[i] = (9679 * R[i-1] + 12637 * i) % 2287
```

Determine the number of iterations required for each of the lists, `L`, `D`, and `R`. Give an approximate formula for how many iterations `quicksort` will need in the case of list `D` from Exercise 1, if `D` has n elements. Explain the difference between the number of iterations required for `D` versus `R`.

Hint: Think of what the sizes of the split lists are in the two cases.

Puzzle Exercise 3: A related problem to sorting is the problem of finding the k^{th} smallest element in an unsorted array. We will assume all elements are distinct to avoid the question of what we mean by the k^{th} smallest when we have repeated elements. One way to solve this problem is to sort and then output the k^{th} element, but we would like something faster.

Notice that in Quicksort, after the partitioning step, we can tell which sublist has the item we are looking for, just by looking at their sizes. So, we only need to recursively examine *one* sublist, not two. For instance, if we are looking for the 17th-smallest element in our list, and after partitioning the sublist of elements less than the pivot, call it `LESS`, has size 100, we then just need to find the 17th smallest element in `LESS`. If the `LESS` sublist has size exactly 16 then we just return the pivot. On the other hand, if the `LESS` sublist has size 10, then we need to look for the 17th smallest element of the original list in `GREATER`, which is the sublist containing elements greater than the pivot.

Modify `quicksort` to code `quickselect` as described above.

Hint: You will not need to modify k in either recursive call, nor do you need to modify `pivotPartitionClever`.